目錄

http/2中文版

根据rfc7540翻译

超文本传输协议（HTTP）是一个非常成功的协议。然而，HTTP/1.1所采用的底层传输方法有几个特征，这些特征影响了应用的性能表现。

例如，HTTP/1.0在指定的TCP连接上同一时刻只允许执行一个请求。虽然HTTP/1.1增加了管道，但是这只部分解决了请求并发的问题，却仍然为报头阻塞所困。因此，HTTP/1.0和HTTP/1.1协议的客户端只能依靠和服务器建立多个连接的策略来实现并发效果，降低时延。

此外，HTTP头部字段经常重复和冗余，引起了不必要的网络拥塞，并且快速填满了初始的TCP拥塞窗口。当多个请求在一个TCP连接上执行时，这将导致过长的的时延。

HTTP/2通过优化HTTP语义传输解决了这些问题。特别地，HTTP/2允许请求和响应信息在同一个连接上交错传递，并且针对HTTP头部提供了一套高效的编码规则。同时，它允许给请求赋予优先级，让重要的请求优先完成，从而进一步提升性能。

由此产生的协议对网络更友好，因为相较于HTTP/1.x，它需要更少的TCP连接。这意味着更少的连接资源竞争，更长的连接生命周期。这些反过来提高了网络的利用率。

最后，HTTP/2允许使用二进制消息帧以提高消息的处理效率。

HTTP/2提供了HTTP语义的传输优化。HTTP/2支持HTTP/1.1的所有核心特征，并且在很多方面做的更高效。

HTTP/2的基本协议单元是帧（第4章）。每个帧类型都有不同的目的和用途。例如，报头帧（HEADERS）和数据帧（DATA）组成了基本的HTTP请求和响应；其他的帧类型例如设置帧（SETTINGS）、窗口更新帧（WINDOW_UPDATE）、推送承诺帧（PUSH_PROMISE）被用来支持HTTP/2的其他特性。

请求多路复用是通过将HTTP请求-响应交换关联到一个流上来实现的。因为流相互之间是相互独立的，所以一个流上的请求或响应发生阻塞或停止不会影响其他流。

Flow control and prioritization ensure that it is possible to efficiently use multiplexed streams. Flow control (Section 5.2) helps to ensure that only data that can be used by a receiver is transmitted. Prioritization (Section 5.3) ensures that limited resources can be directed to the most important streams first.

流量控制和优先级保证了可以高效的使用的多路复用流。流量控制有助于保证只有可以被接受者使用的数据才会被传输。优先级使得最重要的流可以优先使用有限的资源。

HTTP/2 adds a new interaction mode, whereby a server can push responses to a client (Section 8.2). Server push allows a server to speculatively send data to a client that the server anticipates the client will need, trading off some network usage against a potential latency gain. The server does this by synthesizing a request, which it sends as a PUSH_PROMISE frame. The server is then able to send a response to the synthetic request on a separate stream.

HTTP/2添加了一种新的交互模式，这种交互模式中服务器可以向客户端推送响应。服务器推送允许服务器根据预测客户端可能需要的数据，主动向客户端发送数据，用一点网络的使用来交换潜在的时延降低。服务器通过合成一个通过PUSH_PROMISE帧发送的请求来实现推送。然后服务器就可以在一个单独的流里面响应这个合成请求。

Because HTTP header fields used in a connection can contain large amounts of redundant data, frames that contain them are compressed (Section 4.3). This has especially advantageous impact upon request sizes in the common case, allowing many requests to be compressed into one packet.

因为一个连接中的HTTP报头字段可能包含大量的冗余数据，所以包含报头字段的帧是被压缩的。通常情况下压缩能够极大地减少请求大小，从而实现将多个请求放入一个包里。

The HTTP/2 specification is split into four parts:

- Starting HTTP/2 (Section 3) covers how an HTTP/2 connection is initiated.
- The framing (Section 4) and streams (Section 5) layers describe the way HTTP/2 frames are structured and formed into multiplexed streams.
- Frame (Section 6) and error (Section 7) definitions include details of the frame and error types used in HTTP/2.
- HTTP mappings (Section 8) and additional requirements (Section 9) describe how HTTP semantics are expressed using frames and streams.

HTTP/2规范分为一下四个部分：

- 启动HTTP/2包含了一个HTTP/2连接是如何初始化的。
- 分帧和流层描述了HTTP/2的帧是如何构成复用流的。
- 帧和错误定义包括了HTTP/2中使用的帧和错误类型的详细内容。
- HTTP寻址和附加需求描述了如何用帧和流表达HTTP语义。

While some of the frame and stream layer concepts are isolated from HTTP, this specification does not define a completely generic framing layer. The framing and streams layers are tailored to the needs of the HTTP protocol and server push.

因为一些帧和流的概念是和HTTP分离的，这个规范没有定义一个完全通用的帧层。帧和流层是根据HTTP协议和服务器推送的需要定制的。

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

文档中出现的关键字“必须”,“绝对不能”，“要求”，“应”，“不应”，“应该”，“不应该”，“建议”，“可以”及“可选”可通过在 RFC 2119 的解释进行理解【RFC2119】

All numeric values are in network byte order. Values are unsigned unless otherwise indicated. Literal values are provided in decimal or hexadecimal as appropriate. Hexadecimal literals are prefixed with 0x to distinguish them from decimal literals.

所有的数值都是按网络字节顺序。除非有另外说明，数值是无符号的。按情况提供十进制或十六进制的文本值。十六进制用前缀0x来区分。

The following terms are used:

- **client**: The endpoint that initiates an HTTP/2 connection. Clients send HTTP requests and receive HTTP responses.
- **connection:** A transport-layer connection between two endpoints.
- **connection error:** An error that affects the entire HTTP/2 connection.
- **endpoint:** Either the client or server of the connection.
- **frame:** The smallest unit of communication within an HTTP/2 connection, consisting of a header and a variable-length sequence of octets structured according to the frame type.
- **peer:** An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.
- **receiver:** An endpoint that is receiving frames.
- **sender:** An endpoint that is transmitting frames.
- **server:** The endpoint that accepts an HTTP/2 connection. Servers receive HTTP requests and serve HTTP responses.
- **stream:** A bi-directional flow of frames within the HTTP/2 connection.
- **stream error:** An error on the individual HTTP/2 stream.

文中使用到的术语包括：

- 客户端：发起HTTP/2请求的端点。客户端发送请求接受响应。
- 连接：在两个端点之间的传输层级别的连接。
- 连接错误：影响整个HTTP/2连接的错误。

- 端点：连接的客户端或服务器。
- 帧：HTTP/2.0通信连接中的最小单元，包括根据帧类型结构的字节的报头和可变长度 的序列。
- 对等端：一个端点。当讨论特定的端点时，“对等端”指的是讨论的主题的远程端点。
- 接收端：正在接收帧的端点。
- 发送端：正在传输帧的端点。
- 服务端：接受HTTP/2连接的端点。服务器接受请求发送响应。
- 流：HTTP/2连接中的一个双向字节帧流。
- 流错误：一个HTTP/2流中的错误

Finally, the terms "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 2.3 of [RFC7230]. Intermediaries act as both client and server at different times.

The term "payload body" is defined in Section 3.3 of [RFC7230].

最后，文档[RFC7230]2.3节给出了术语“网关”、“中间层”、“代理”和“隧道”的定义。中间层在不同时间可以既做客户端又可以做服务器。

文档[RFC7230]3.3节给出了术语“载体”的定义。

An HTTP/2 connection is an application-layer protocol running on top of a TCP connection ([TCP]). The client is the TCP connection initiator.

HTTP/2连接是运行在TCP连接上的应用层协议。客户端是TCP连接的发起者。

HTTP/2 uses the same "http" and "https" URI schemes used by HTTP/1.1. HTTP/2 shares the same default port numbers: 80 for "http" URIs and 443 for "https" URIs. As a result, implementations processing requests for target resource URIs like http://example.org/foo or https://example.com/bar are required to first discover whether the upstream server (the immediate peer to which the client wishes to establish a connection) supports HTTP/2.

HTTP/2使用和HTTP/1.1一样的“http”和“https” URI。HTTP也使用同样的默认端口：“http”的80和“https”的443。因此，请求类似http://example.org/foo 或 https://example.com/bar的URI前需要先判断上游(和客户端直接相连的端，可以是服务器或代理中介)是否支持HTTP/2。

The means by which support for HTTP/2 is determined is different for "http" and "https" URIs. Discovery for "http" URIs is described in Section 3.2. Discovery for "https" URIs is described in Section 3.3.

“http" URI和“https” URI判断上游是否支持HTTP/2的方法是不一样的。“http” URI的判断方法在3.2节描述。“https” URI的判断方法在3.3描述。

The protocol defined in this document has two identifiers.

- The string "h2" identifies the protocol where HTTP/2 uses Transport Layer Security (TLS) [TLS12]. This identifier is used in the TLS application-layer protocol negotiation (ALPN) extension [TLS-ALPN] field and in any place where HTTP/2 over TLS is identified.

  The "h2" string is serialized into an ALPN protocol identifier as the two-octet sequence: 0x68, 0x32.

- The string "h2c" identifies the protocol where HTTP/2 is run over cleartext TCP. This identifier is used in the HTTP/1.1 Upgrade header field and in any place where HTTP/2 over TCP is identified.

  The "h2c" string is reserved from the ALPN identifier space but describes a protocol that does not use TLS.

Negotiating "h2" or "h2c" implies the use of the transport, security, framing, and message semantics described in this document.

本文定义的协议有两个标示符。

- 字符串"h2"表示HTTP/2协议使用TLS[TLS12]。这个标示符用在TLS-ALPN字段中和任何基于TLS的HTTP/2协议中。

  字符串"h2"以两个十六进制序列：0x86, 0x32形式序列化到ALPN协议标示符中。

- 字符串"h2c"表示HTTP/2协议运行在明文TCP上。这个标示符用在HTTP/1.1升级报头字段中以及任何基于TCP的HTTP/2协议中。

  "h2c"是从ALPN标示符中保留下来的，但是它描述的是不适用TLS的协议。

讨论"h2"或者"h2c"意味着使用本文描述的传输、安全、帧和消息语义。

A client that makes a request for an "http" URI without prior knowledge about support for HTTP/2 on the next hop uses the HTTP Upgrade mechanism (Section 6.7 of [RFC7230]). The client does so by making an HTTP/1.1 request that includes an Upgrade header field with the "h2c" token. Such an HTTP/1.1 request MUST include exactly one HTTP2-Settings (Section 3.2.1) header field.

For example:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

在不知道下一跳是否支持HTTP/2的情况下，客户端使用HTTP升级机制来向“http” URI发送请求（[RFC7230]6.7节）。客户端构造一个升级报头字段值为"h2c"的请求来做到这一点。这样的HTTP/1.1请求必须且只能包含一个HTTP2-Settings头部字段。

例如：

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

Requests that contain a payload body MUST be sent in their entirety before the client can send HTTP/2 frames. This means that a large request can block the use of the connection until it is completely sent.

客户端必须在全部发送完包含主体内容的请求之后才能发送HTTP/2帧。这意味着一个大的请求在完全发送完之前可以阻塞连接。

If concurrency of an initial request with subsequent requests is important, an OPTIONS request can be used to perform the upgrade to HTTP/2, at the cost of an additional round trip.

如果一个初始请求后的后续请求的并发性很重要，则可以用一个额外的请求将协议来升级到HTTP/2，代价是多一个来回的时间。

A server that does not support HTTP/2 can respond to the request as though the Upgrade header field were absent:

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

...
```

不支持HTTP/2的服务器也可以回复升级请求，只不过响应不包括升级报头字段，例子：

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

...
```

A server MUST ignore an "h2" token in an Upgrade header field. Presence of a token with "h2" implies HTTP/2 over TLS, which is instead negotiated as described in Section 3.3.

服务器必须忽略升级报头中的"h2"。"h2"的存在意味着这是基于TLS实现的HTTP/2，这部分内容将在3.3节讨论。

A server that supports HTTP/2 accepts the upgrade with a 101 (Switching Protocols) response. After the empty line that terminates the 101 response, the server can begin sending HTTP/2 frames. These frames MUST include a response to the request that initiated the upgrade.

For example:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

一个支持HTTP/2的服务器接受升级请求并返回响应101（切换协议）。在发送空行表示101响应结束后，服务器就可以开始发送HTTP/2帧了。这些帧必须包含对发起升级的请求的响应。

例如：

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

The first HTTP/2 frame sent by the server MUST be a server connection preface (Section 3.5) consisting of a SETTINGS frame (Section 6.5). Upon receiving the 101 response, the client MUST send a connection preface (Section 3.5), which includes a SETTINGS frame.

服务器发送的第一个HTTP/2帧必须是包含一个SETTINGS帧的服务器连接前言（3.5节）。一旦收到101响应，客户端也必须发送一个包含SETTINGS帧的连接前言。

The HTTP/1.1 request that is sent prior to upgrade is assigned a stream identifier of 1 (see Section 5.1.1) with default priority values (Section 5.3.5). Stream 1 is implicitly "half-closed" from the client toward the server (see Section 5.1), since the request is completed as an HTTP/1.1 request. After commencing the HTTP/2 connection, stream 1 is used for the response.

在升级之前发送的HTTP/1.1的请求被分配一个标示为1的流并赋予默认优先级。流1是从客户端到服务器是隐式半封闭的，因为这个请求是按照HTTP/1.1协议请求完成的。在HTTP/2连接完成后，流1将被用来发送响应。

A request that upgrades from HTTP/1.1 to HTTP/2 MUST include exactly one HTTP2-Settings header field. The HTTP2-Settings header field is a connection-specific header field that includes parameters that govern the HTTP/2 connection, provided in anticipation of the server accepting the request to upgrade.

```
HTTP2-Settings    = token68
```

升级HTTP/1.1到HTTP/2的升级请求必须有且只能有一个HTTP2-Settings报头字段。HTTP2-Settings报头字段是一个连接特定报头字段，它包含控制HTTP/2连接的参数，提供该参数期望服务器能够接受升级请求。

A server MUST NOT upgrade the connection to HTTP/2 if this header field is not present or if more than one is present. A server MUST NOT send this header field.

如果HTTP2-Settings报头字段没有提供或者提供了不止一个，服务器不应该升级连接到HTTP/2。另外，服务器不能发送这个报头字段。

The content of the HTTP2-Settings header field is the payload of a SETTINGS frame (Section 6.5), encoded as a base64url string (that is, the URL- and filename-safe Base64 encoding described in Section 5 of [RFC4648], with any trailing '=' characters omitted). The ABNF [RFC5234] production for token68 is defined in Section 2.1 of [RFC7235].

HTTP2-Settings报头字段的内容是SETTINGS帧的载体，被编码成base64url格式的字符串（即，[RFC4648]第五节中的对URL和文件名安全的Base64编码，忽略任何“=”字符）。ABNF[RFC5234]产品中对token68的定义在[RFC7235]2.1节中。

Since the upgrade is only intended to apply to the immediate connection, a client sending the HTTP2-Settings header field MUST also send HTTP2-Settings as a connection option in the Connection header field to prevent it from being forwarded (see Section 6.1 of [RFC7230]).

因为升级是对当前连接而言的，所以发送HTTP2-Settings报头字段的客户端也必须在Connection报头字段中发送字符串"HTTP2-Settings"来防止请求被转发。例如：

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

A server decodes and interprets these values as it would any other SETTINGS frame. Explicit acknowledgement of these settings (Section 6.5.3) is not necessary, since a 101 response serves as implicit acknowledgement. Providing these values in the upgrade

request gives a client an opportunity to provide parameters prior to receiving any frames from the server.

服务器像对任何其他SETTINGS帧一样解码和解释这些值。因为101响应是作为一个隐式确认，所以对没有必要对这些设置进行显示确认。在升级请求中提供这些值赋予了客户端在收到任何来自服务器的帧之前提供这些参数的机会。

A client that makes a request to an "https" URI uses TLS [TLS12] with the application-layer protocol negotiation (ALPN) extension [TLS-ALPN].

客户端利用TLS和应用层协议协商扩展构建https请求。

HTTP/2 over TLS uses the "h2" protocol identifier. The "h2c" protocol identifier MUST NOT be sent by a client or selected by a server; the "h2c" protocol identifier describes a protocol that does not use TLS.

基于TLS的HTTP/2使用“h2”作为协议标识符。“h2c”协议标识符不能被客户端不能发送或者被服务器选择；“h2c”协议标识符表示该协议不使用TLS。

Once TLS negotiation is complete, both the client and the server MUST send a connection preface (Section 3.5).

当TLS协商完成后，客户端和服务器端都必须发送一个连接前言。

A client can learn that a particular server supports HTTP/2 by other means. For example, [ALT-SVC] describes a mechanism for advertising this capability.

客户端可以通过其他方式获得特定服务器是否支持HTTP/2。例如，[ALT-SVC]就描述了一种机制来广播这种能力（支持HTTP/2）。

A client MUST send the connection preface (Section 3.5) and then MAY immediately send HTTP/2 frames to such a server; servers can identify these connections by the presence of the connection preface. This only affects the establishment of HTTP/2 connections over cleartext TCP; implementations that support HTTP/2 over TLS MUST use protocol negotiation in TLS [TLS-ALPN].

客户端必须向服务器发送连接前言，同时可以立即发送HTTP/2序列帧；服务器根据连接前言鉴别这些连接。这只对基于明文TCP协议的HTTP/2连接有效；基于TLS的HTTP/2连接必须使用TLS的协议协商。

Likewise, the server MUST send a connection preface (Section 3.5).

类似的，服务器也必须发送一个连接前言。

Without additional information, prior support for HTTP/2 is not a strong signal that a given server will support HTTP/2 for future connections. For example, it is possible for server configurations to change, for configurations to differ between instances in clustered servers, or for network conditions to change.

没有额外的信息的情况下，之前对HTTP/2的支持并不能表明这个服务器在未来的连接中也支持HTTP/2。例如，可能服务器配置发生了变化，可能集群服务器的配置各不相同，或者网络环境发生了改变。

In HTTP/2, each endpoint is required to send a connection preface as a final confirmation of the protocol in use and to establish the initial settings for the HTTP/2 connection. The client and server each send a different connection preface.

在HTTP/2协议中，每个端都需要发送一个连接前言作为对协议使用的最终确认并确定HTTTP/2连接的初始设置参数。

The client connection preface starts with a sequence of 24 octets, which in hex notation is:

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

客户端连接前言是以24字节的序列开始的，以十六进制表示是:

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

That is, the connection preface starts with the string PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n). This sequence MUST be followed by a SETTINGS frame (Section 6.5), which MAY be empty. The client sends the client connection preface immediately upon receipt of a 101 (Switching Protocols) response (indicating a successful upgrade) or as the first application data octets of a TLS connection. If starting an HTTP/2 connection with prior knowledge of server support for the protocol, the client connection preface is sent upon connection establishment.

即，这个连接前言以字符串 PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n 开始。这个序列后必须紧跟着一个SETTINGS帧，不过SETTINGS帧可以为空。一旦客户端收到101（协议转换）响应（表示协议升级成功）就立即发送连接前言，或者作为TLS连接的第一个应用层数据字节。如果事先知道服务器支持HTTP/2的情况下启动一个HTTP/2连接，则客户端连接前言在连接建立后发送。

- Note: The client connection preface is selected so that a large proportion of HTTP/1.1 or HTTP/1.0 servers and intermediaries do not attempt to process further frames. Note that this does not address the concerns raised in [TALKING].
- 注意：客户端连接前言的存在是为了让很大比例的HTTP/1.1和HTTP/1.0协议的服务器或中间节点不试图去处理接下来的帧数据。注意这并没有解决[讨论]中提出的问题。

The server connection preface consists of a potentially empty SETTINGS frame (Section 6.5) that MUST be the first frame the server sends in the HTTP/2 connection.

服务器连接前言包含一个可能为空的SETTINGS帧，它必须是HTTP/2连接中服务器发送的第一个帧。

The SETTINGS frames received from a peer as part of the connection preface MUST be acknowledged (see Section 6.5.3) after sending the connection preface.

从另一方收到的连接前言中的SETTINGS帧必须在发送连接前言后进行确认。（例如客户端收到服务器连接前言中的SETTINGS帧，则必须在发送客户端连接前言后对这个SETTINGS帧进行确认）

To avoid unnecessary latency, clients are permitted to send additional frames to the server immediately after sending the client connection preface, without waiting to receive the server connection preface. It is important to note, however, that the server connection preface SETTINGS frame might include parameters that necessarily alter how a client is expected to communicate with the server. Upon receiving the SETTINGS frame, the client is expected to honor any parameters established. In some configurations, it is possible for the server to transmit SETTINGS before the client sends additional frames, providing an opportunity to avoid this issue.

为了减少不必要的时延，允许客户端发送客户端连接前言时同时发送额外的帧，而不用等收到服务器连接前言后再发送。然而，必须注意到：服务器连接前言的SETTINS帧可能包含规定客户端如何与服务器通信的参数。一旦收到SETTINGS帧，客户端就应当遵守这些参数的设定。在某些配置下，服务器可以在客户端发送额外的帧之前发送SETTINGS帧，从而避免上面的问题。

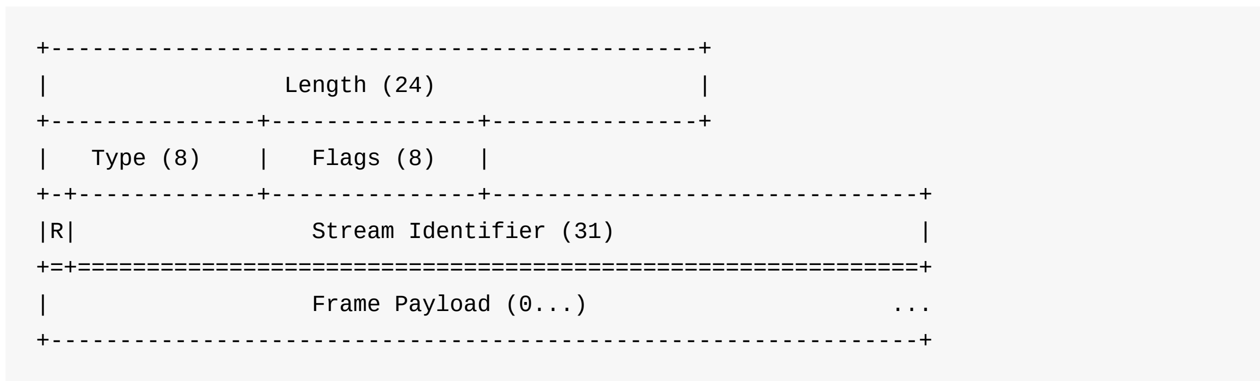
Clients and servers MUST treat an invalid connection preface as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. A GOAWAY frame (Section 6.8) MAY be omitted in this case, since an invalid preface indicates that the peer is not using HTTP/2.

服务器和客户端必须将错误的连接前言看作是PROTOCOL_ERROR类型的连接错误。在这种情况下，GOAWAY帧可以忽略，因为一个非法的连接前言意味着连接并非使用HTTP/2。

Once the HTTP/2 connection is established, endpoints can begin exchanging frames.

一旦HTTP/2连接建立完成，连接的两端就可以开始交换帧了。

All frames begin with a fixed 9-octet header followed by a variable-length payload. 每个帧都是以9个字节的头部开始的，后面接着可变长度的数据。



```
+-----------------------------------------------+
|                 Length (24)                   |
+---------------+---------------+---------------+
|   Type (8)    |   Flags (8)   |
+-+-------------+---------------+-------------------------------+
|R|                 Stream Identifier (31)                      |
+=+=============================================================+
|                   Frame Payload (0...)                      ...
+---------------------------------------------------------------+
```

Figure 1: Frame Layout

The fields of the frame header are defined as:

帧头部字段定义如下：

Length: The length of the frame payload expressed as an unsigned 24-bit integer. Values greater than 2^14 (16,384) MUST NOT be sent unless the receiver has set a larger value for SETTINGS_MAX_FRAME_SIZE.

The 9 octets of the frame header are not included in this value.

帧的数据长度用一个24位的整数表示。但是，除非接收者给SETTINGS_MAX_FRAME_SIZE设置了更大的值，否则数据长度不要超过2^14 (16384)字节。

头部的9字节不算在这个长度里。

Type: The 8-bit type of the frame. The frame type determines the format and semantics of the frame. Implementations MUST ignore and discard any frame that has a type that is unknown.

接下来的8比特表示帧类型。帧类型决定了帧的格式和语义。协议实现者必须忽略并丢弃类型为unknown类型的帧。

Flags: An 8-bit field reserved for boolean flags specific to the frame type.

Flags are assigned semantics specific to the indicated frame type. Flags that have no defined semantics for a particular frame type MUST be ignored and MUST be left unset (0x0) when sending.

有8个比特是为帧类型相关的布尔标识预留的。

标识对于不同的帧类型赋予了不同的语义。如果标识对于某种帧类型没有定义语义，则它必须被忽略且发送的时候应该赋值为(0x0)。

R: A reserved 1-bit field. The semantics of this bit are undefined, and the bit MUST remain unset (0x0) when sending and MUST be ignored when receiving.

R是一个保留的比特字段。这个比特的语义没有定义，发送时它必须被设置为(0x0), 接收时需要忽略。

Stream Identifier: A stream identifier (see Section 5.1.1) expressed as an unsigned 31-bit integer. The value 0x0 is reserved for frames that are associated with the connection as a whole as opposed to an individual stream.

The structure and content of the frame payload is dependent entirely on the frame type.

流标识用31比特的无符号整数表示。值0x0是保留的，表示帧是于与连接作为整体的，而不是单独的流。

帧负载的内容和结构完全依赖于帧类型。

The size of a frame payload is limited by the maximum size that a receiver advertises in the SETTINGS_MAX_FRAME_SIZE setting. This setting can have any value between 2^14 (16,384) and 2^24-1 (16,777,215) octets, inclusive.

帧的有效负载的大小受到接收者广播的SETTINGS_MAX_FRAME_SIZE限制。大小可以是2^14 (16,384)字节到2^24-1 (16,777,215)字节之间的任意值。

All implementations MUST be capable of receiving and minimally processing frames up to 2^14 octets in length, plus the 9-octet frame header (Section 4.1). The size of the frame header is not included when describing frame sizes.

所有的协议实现必须能够接收和处理长度大于等于2^14字节的帧，加上9字节的帧头部。帧长度不包括帧头部的长度。

- Note: Certain frame types, such as PING (Section 6.7), impose additional limits on the amount of payload data allowed.
- 注意：对于特定类型的帧，例如PING帧，可以给负载长度设置另外的大小限制。

An endpoint MUST send an error code of FRAME_SIZE_ERROR if a frame exceeds the size defined in SETTINGS_MAX_FRAME_SIZE, exceeds any limit defined for the frame type, or is too small to contain mandatory frame data. A frame size error in a frame that could alter the state of the entire connection MUST be treated as a connection error (Section 5.4.1); this includes any frame carrying a header block (Section 4.3) (that is, HEADERS, PUSH_PROMISE, and CONTINUATION), SETTINGS, and any frame with a stream identifier of 0.

如果一个帧大小超过了SETTINGS_MAX_FRAME_SIZE，或者超过了对应类型的帧的长度限制，或者达不到强制的帧数据的最小长度，端点必须发送错误码FRAME_SIZE_ERROR。一个能够影响整个连接状态的帧大小错误必须按照连接错误来对待。包括任何携带头部块（即，HEADERS, PUSH_PROMISE, and CONTINUATION），设置帧，以及任何流id为0的帧。

Endpoints are not obligated to use all available space in a frame. Responsiveness can be improved by using frames that are smaller than the permitted maximum size. Sending large frames can result in delays in sending time-sensitive frames (such as RST_STREAM, WINDOW_UPDATE, or PRIORITY), which, if blocked by the transmission of a large frame, could affect performance.

端不必使用帧的所有可用空间。通过使用大小小于最大限制的帧可以提高响应性能（即推荐使用小一点的帧）。发送大的帧可能延误时间敏感型的帧（例如RST_STREAM, WINDOW_UPDATE, 和 PRIORITY帧）的发送，这些帧的发送如果被大帧的传输阻塞，则会影响性能表现。

Just as in HTTP/1, a header field in HTTP/2 is a name with one or more associated values. Header fields are used within HTTP request and response messages as well as in server push operations (see Section 8.2).

和HTTP/1一样，HTTP/2中的头部也是一个名字关联着一个或多个值。头部字段被使用在HTTP请求和响应，以及服务器推送操作中。

Header lists are collections of zero or more header fields. When transmitted over a connection, a header list is serialized into a header block using HTTP header compression [COMPRESSION]. The serialized header block is then divided into one or more octet sequences, called header block fragments, and transmitted within the payload of HEADERS (Section 6.2), PUSH_PROMISE (Section 6.6), or CONTINUATION (Section 6.10) frames.

头部列表是0个或多个头部字段的集合。在传输的时候，头部列表使用HTTP头部压缩算法序列化到头部块。然后序列化的头部块被切分成一个或多个字节序列，称为头部块帧，然后作为HEADERS、PUSH_PROMISE或CONTINUATION帧的负载传输。

The Cookie header field [COOKIE] is treated specially by the HTTP mapping (see Section 8.1.2.5).

Cookie报头字段被HTTP映射特殊处理。

A receiving endpoint reassembles the header block by concatenating its fragments and then decompresses the block to reconstruct the header list.

接收端通过串联帧片段来重新组合头部块，然后解压头部块重构头部列表。

A complete header block consists of either:

- a single HEADERS or PUSH_PROMISE frame, with the END_HEADERS flag set, or
- a HEADERS or PUSH_PROMISE frame with the END_HEADERS flag cleared and one or more CONTINUATION frames, where the last CONTINUATION frame has the END_HEADERS flag set.

一个完整的头部包括了：

- 一个单独的设置了END_HEADERS标识的HEADERS帧或PUSH_PROMISE帧，或者
- 一个没有设置END_HEADERS标识的HEADERS帧或PUSH_PROMISE帧加上一个或多个CONTINUATION帧，并且最后一个CONTINUATION帧设置了END_HEADERS标识。

Header compression is stateful. One compression context and one decompression context are used for the entire connection. A decoding error in a header block MUST be treated as a connection error (Section 5.4.1) of type COMPRESSION_ERROR.

头部压缩是有状态的。整个连接都必须一致使用压缩或者一致不使用压缩。头部的解码错误必须按照COMPRESSION_ERROR类型的连接错误来处理。

Each header block is processed as a discrete unit. Header blocks MUST be transmitted as a contiguous sequence of frames, with no interleaved frames of any other type or from any other stream. The last frame in a sequence of HEADERS or CONTINUATION frames has the END_HEADERS flag set. The last frame in a sequence of PUSH_PROMISE or CONTINUATION frames has the END_HEADERS flag set. This allows a header block to be logically equivalent to a single frame.

每个头部块是离散处理的。头部可必须按照连续的序列帧来传送，中间不能夹杂任何其他流的或者其他类型的帧。HEADERS和CONTINUATION帧序列中最一个必须设置了END_HEADERS标识。PUSH_PROMISE和CONTINUATION帧序列中最一个也必须设置了END_HEADERS标识。这样使得头部块逻辑等同于一个帧。

Header block fragments can only be sent as the payload of HEADERS, PUSH_PROMISE, or CONTINUATION frames because these frames carry data that can modify the compression context maintained by a receiver. An endpoint receiving HEADERS, PUSH_PROMISE, or CONTINUATION frames needs to reassemble header blocks and perform decompression even if the frames are to be discarded. A receiver MUST terminate the connection with a connection error (Section 5.4.1) of type COMPRESSION_ERROR if it does not decompress a header block.

头部分片只能由HEADERS，PUSH_PROMISE或CONTINUATION的载体中携带，因为这些帧可以修改接收者的压缩环境设置。一个端收到HEADERS，PUSH_PROMISE或CONTINUATION帧后，需要重组头部块并解压缩，即使这些帧最终被废弃。如果接收者无法解压缩头部，那么必须报COMPRESSION_ERROR类型的连接错误并终止连接。

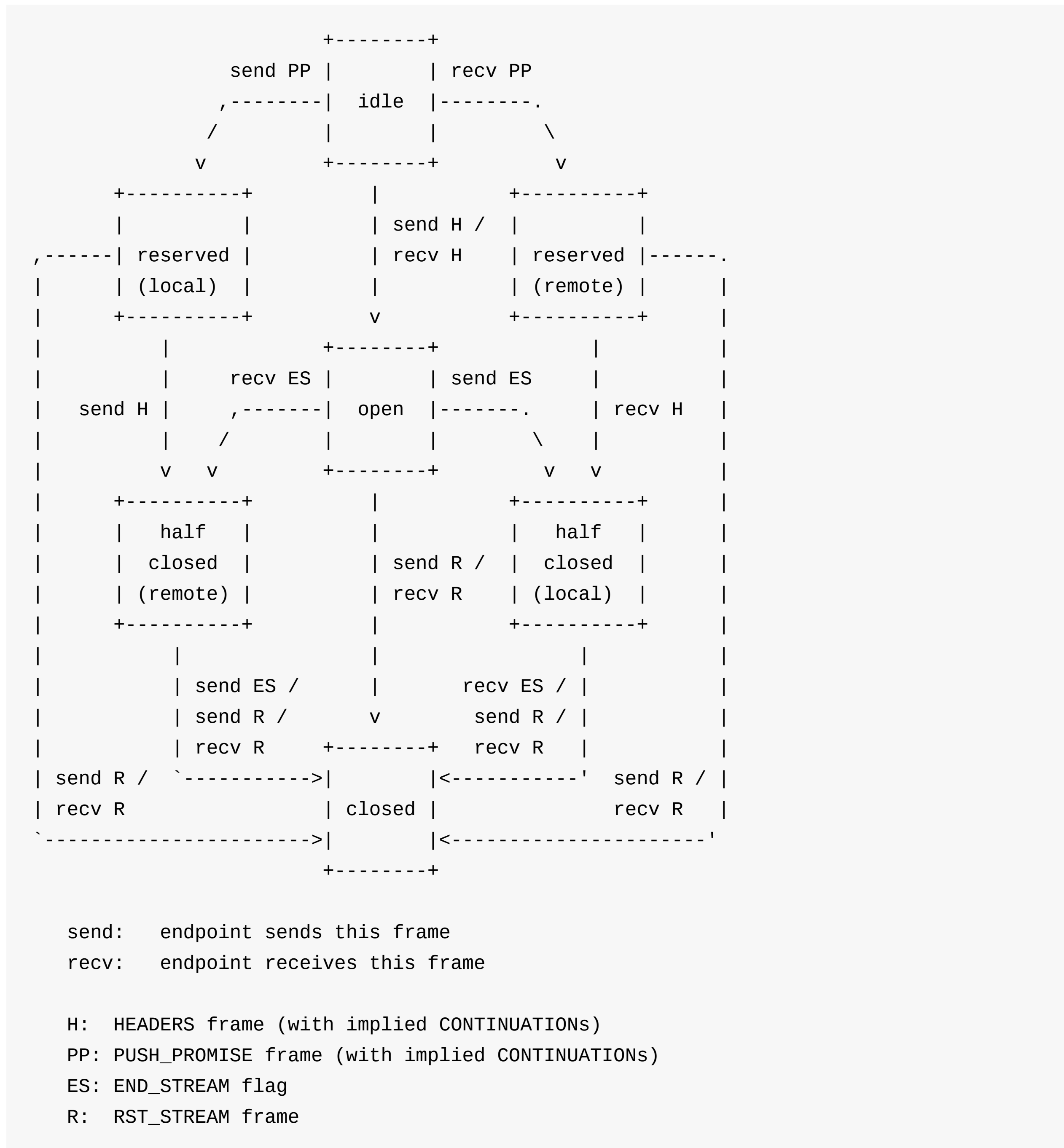
A "stream" is an independent, bidirectional sequence of frames exchanged between the client and server within an HTTP/2 connection. Streams have several important characteristics:

- A single HTTP/2 connection can contain multiple concurrently open streams, with either endpoint interleaving frames from multiple streams.
- Streams can be established and used unilaterally or shared by either the client or server.
- Streams can be closed by either endpoint.
- The order in which frames are sent on a stream is significant. Recipients process frames in the order they are received. In particular, the order of HEADERS and DATA frames is semantically significant.
- Streams are identified by an integer. Stream identifiers are assigned to streams by the endpoint initiating the stream.

流是服务器和客户端在HTTP/2连接里进行交换的一个独立、双工的帧序列。流具有若干重要特征：

- 一个单独的HTTP/2连接可以包含多个并发的流，任何一个端都可以同时处理来自多个流的交错帧。
- 流可以被客户端或者服务器单独建立和使用。
- 流可以被任何一端关闭。
- 流中的帧的发送顺序是非常重要的。接收者按照帧的接受顺序来处理帧。特别地，DATA和HEADERS帧的顺序是非常重要的。
- 流依靠一个整数进行标识。流标识由建立流的端赋予。

The lifecycle of a stream is shown in Figure 2. 图2显示了流的生命周期。



```
                         +--------+
                 send PP |        | recv PP
                ,--------|  idle  |--------.
               /         |        |         \
              v          +--------+          v
       +----------+          |           +----------+
       |          |          | send H /  |          |
,------| reserved |          | recv H    | reserved |------.
|      | (local)  |          |           | (remote) |      |
|      +----------+          v           +----------+      |
|          |             +--------+             |          |
|          |     recv ES |        | send ES     |          |
|   send H |     ,-------|  open  |-------.     | recv H   |
|          |    /        |        |        \    |          |
|          v   v         +--------+         v   v          |
|      +----------+          |           +----------+      |
|      |   half   |          |           |   half   |      |
|      |  closed  |          | send R /  |  closed  |      |
|      | (remote) |          | recv R    | (local)  |      |
|      +----------+          |           +----------+      |
|           |                |                 |           |
|           | send ES /      |       recv ES / |           |
|           | send R /       v        send R / |           |
|           | recv R     +--------+   recv R   |           |
| send R /  `----------->|        |<-----------'  send R / |
| recv R                 | closed |               recv R   |
`----------------------->|        |<----------------------'
                         +--------+

   send:   endpoint sends this frame
   recv:   endpoint receives this frame

   H:  HEADERS frame (with implied CONTINUATIONs)
   PP: PUSH_PROMISE frame (with implied CONTINUATIONs)
   ES: END_STREAM flag
   R:  RST_STREAM frame
```

Figure 2: Stream States

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. In this regard, CONTINUATION frames do not result in state transitions; they are effectively part of the HEADERS or PUSH_PROMISE that they follow. For the purpose of state transitions, the END_STREAM flag is processed as a separate event to the frame that bears it; a HEADERS frame with the END_STREAM flag set can cause two state transitions.

上面的图显示了流的状态转换以及导致状态转换的帧和标记。虽然CONTINUATION帧并不会引起流的状态转换，单它是他所紧随的HEADERS帧或PUSH_PROMISE帧的有效的组成部分。为了便于描述状态转移，END_STREAM的标识在处理的时候特意和承载它的帧区分开；所以一个带有END_STREAM标识的HEADERS帧会导致两个状态转移。

Both endpoints have a subjective view of the state of a stream that could be different when frames are in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. The negative consequences of a mismatch in states are limited to the "closed" state after sending RST_STREAM, where frames might be received for some time after closing.

两个端点都对流的状态有一个主观的视图，这个视图在帧眞在传输的时候可能会不一样。在创建流的的时候端点之间不需要相互协调，任何一个端点都可以单方面的创建流。状态不匹配的负面后果仅限于发送RST_STREAM之后的关闭状态，有时候可能会在关闭之后收到帧。

Streams have the following states:

流具有以下状态：

idle:

All streams start in the "idle" state.

The following transitions are valid from this state:

- Sending or receiving a HEADERS frame causes the stream to become "open". The stream identifier is selected as described in Section 5.1.1. The same HEADERS frame can also cause a stream to immediately become "half-closed".
- Sending a PUSH_PROMISE frame on another stream reserves the idle stream that is identified for later use. The stream state for the reserved stream transitions to "reserved (local)".
- Receiving a PUSH_PROMISE frame on another stream reserves an idle stream that is identified for later use. The stream state for the reserved stream transitions to "reserved (remote)".
- Note that the PUSH_PROMISE frame is not sent on the idle stream but references the newly reserved stream in the Promised Stream ID field.

Receiving any frame other than HEADERS or PRIORITY on a stream in this state MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

空闲：

所有的流都是以空闲状态开始的。

空闲状态下，以下的状态转换是合法的：

- 发送或者接受到HEADERS帧之后流状态变为打开。流标识的选择在5.1.1节中描述。相同的HEADERS帧也可以导致流状态变为半关闭状态。
- 在另一个流发送一个PUSH_PROMISE帧将保留这个空闲流供后续使用。这个流的状态转换为“本地预留”。
- 在另一个流收到一个PUSH_PROMISE帧将保留这个空闲流供后续使用。这个流的状态转换为“远程预留”。
- 注意PUSH_PROMISE不是从空闲流发送的，只是在承诺流ID字段中引用了新的预留的流。

在空闲状态下的流收到任何除HEADERS帧或者PRIORITY帧以外帧都必须处理为协议错误类型的连接错误。

reserved (local):

A stream in the "reserved (local)" state is one that has been promised by sending a PUSH_PROMISE frame. A PUSH_PROMISE frame reserves an idle stream by associating the stream with an open stream that was initiated by the remote peer (see Section 8.2). 半关闭（“”）”

In this state, only the following transitions are possible:

- The endpoint can send a HEADERS frame. This causes the stream to open in a "half-closed (remote)" state.”。“”
- Either endpoint can send a RST_STREAM frame to cause the stream to become "closed". This releases the stream reservation.

An endpoint MUST NOT send any type of frame other than HEADERS, RST_STREAM, or PRIORITY in this state.

A PRIORITY or WINDOW_UPDATE frame MAY be received in this state. Receiving any type of frame other than RST_STREAM, PRIORITY, or WINDOW_UPDATE on a stream in this state MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

本地预留

一个本地预留状态表示因为发送了一个PUSH_PROMISE帧而被承诺的流状态。PUSH_PROMISE帧通过将空闲流和一个已经被远端初始化的打开的流关联起来实现预留这个空闲流。

在这个状态下，只会发生以下状态转换：

- 端点发送一个HEADERS帧，导致流变成“远程半关闭”状态。
- 任何一端可以发送RST_STREAM帧使得流变为“关闭”状态。这将会释放流的预留。

端点在这个状态下只能发送HEADERS帧、RST_STREAM帧和PRIORITY帧。

这个状态下可以接受PRIORITY、WINDOW_UPDATE和RST_STREAM帧。接收到任何其他帧都应该视为一个协议错误类型的连接错误。

reserved (remote):

A stream in the "reserved (remote)" state has been reserved by a remote peer.

In this state, only the following transitions are possible:

- Receiving a HEADERS frame causes the stream to transition to "half-closed (local)".
- Either endpoint can send a RST_STREAM frame to cause the stream to become "closed". This releases the stream reservation.

An endpoint MAY send a PRIORITY frame in this state to reprioritize the reserved stream. An endpoint MUST NOT send any type of frame other than RST_STREAM, WINDOW_UPDATE, or PRIORITY in this state.

Receiving any type of frame other than HEADERS, RST_STREAM, or PRIORITY on a stream in this state MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

远程预留

一个远程预留状态表示被远程预留了。

在这个状态下，只会发生以下状态转换：

- 收到一个HEADERS镇，流状态变成“本地半关闭”状态。
- 任何一端可以发送RST_STREAM帧使得流变为“关闭”状态。这将会释放流的预留。

这个状态下端点可以发送一个PRIORITY帧来重置预留流的优先级。端点在这个状态下只能发送WINDOW_UPDATE帧、RST_STREAM帧和PRIORITY帧。

这个状态下可以接受PRIORITY、HEADERS和RST_STREAM帧。接收到任何其他帧都应该视为一个协议错误类型的连接错误。

open:

A stream in the "open" state may be used by both peers to send frames of any type. In this state, sending peers observe advertised stream-level flow-control limits (Section 5.2).

From this state, either endpoint can send a frame with an END_STREAM flag set, which causes the stream to transition into one of the "half-closed" states. An endpoint sending an END_STREAM flag causes the stream state to become "half-closed (local)"; an endpoint receiving an END_STREAM flag causes the stream state to become "half-closed (remote)".

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

打开

打开状态下的流可以发送任何帧。在这种状态下，发送端点监控流级别的流量控制限制广播。

任何端点都可以发送一个携带END_STREAM的帧，使得流状态变为某一种“半关闭”状态。发送END_STREAM的端流状态变为“本地半关闭状态”；接受端变为“远程半关闭”状态。

任何一端都可以发送RST_STREAM使得流变为“关闭”状态。

half-closed (local):

A stream that is in the "half-closed (local)" state cannot be used for sending frames other than WINDOW_UPDATE, PRIORITY, and RST_STREAM.

A stream transitions from this state to "closed" when a frame that contains an END_STREAM flag is received or when either peer sends a RST_STREAM frame.

An endpoint can receive any type of frame in this state. Providing flow-control credit using WINDOW_UPDATE frames is necessary to continue receiving flow-controlled frames. In this state, a receiver can ignore WINDOW_UPDATE frames, which might arrive for a short period after a frame bearing the END_STREAM flag is sent.

PRIORITY frames received in this state are used to reprioritize streams that depend on the identified stream.

本地半关闭

处在“本地半关闭”状态的帧不能被用来发送除WINDOW_UPDATE, PRIORITY和RST_STREAM帧以外的帧。

当收到带有END_STREAM标识的帧或者任何一端发送RST_STREAM时，流状态变为“关闭”。

这个状态下端点可以收到任何类型的帧。为了继续收到流量控制帧，利用WINDOW_UPDATE帧提供流量控制授权是非常必要的。在这个状态下，WINDOW_UPDATE帧可能在携带有END_STREAM标识的帧发送之后短暂的时期内到达，而接收者可以忽略这些WINDOW_UPDATE帧。

half-closed (remote):

A stream that is "half-closed (remote)" is no longer being used by the peer to send frames. In this state, an endpoint is no longer obligated to maintain a receiver flow-control window.

If an endpoint receives additional frames, other than WINDOW_UPDATE, PRIORITY, or RST_STREAM, for a stream that is in this state, it MUST respond with a stream error (Section 5.4.2) of type STREAM_CLOSED.

A stream that is "half-closed (remote)" can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream-level flow-control limits (Section 5.2).

A stream can transition from this state to "closed" by sending a frame that contains an END_STREAM flag or when either peer sends a RST_STREAM frame.

远程半关闭

处在“远程半关闭”状态的流不能被对等端用来发送任何帧。在这个状态下，端点不在有义务对接受者流量控制窗口进行维护。

如果在这个状态下，端点收到WINDOW_UPDATE, PRIORITY, 和RST_STREAM帧以外的帧，它必须回复流关闭类型的流错误。

“远程半关闭”的流可以被端点用来发送任何帧。端点继续监控流级别的流量控制限制的广播。

可以通过发送含有END_STREAM标识的帧或者RST_STREAM帧使得流变成“关闭”状态。

closed:

The "closed" state is the terminal state.

关闭

关闭状态是最终状态。

An endpoint MUST NOT send frames other than PRIORITY on a closed stream. An endpoint that receives any frame other than PRIORITY after receiving a RST_STREAM MUST treat that as a stream error (Section 5.4.2) of type STREAM_CLOSED. Similarly, an endpoint that receives any frames after receiving a frame with the END_STREAM flag set MUST treat that as a connection error (Section 5.4.1) of type STREAM_CLOSED, unless the frame is permitted as described below.

端点不能在一个已关闭的流上发除了PRIORITY帧以外的帧。端点收到非PRIORITY帧必须处理为流已关闭错误。同样的，在收到带有END_STREAM帧之后收到任何帧都必须视为流关闭类型的连接错误。除非帧符合以下条件：

WINDOW_UPDATE or RST_STREAM frames can be received in this state for a short period after a DATA or HEADERS frame containing an END_STREAM flag is sent. Until the remote peer receives and processes RST_STREAM or the frame bearing the END_STREAM flag, it might send frames of these types. Endpoints MUST ignore WINDOW_UPDATE or RST_STREAM frames received in this state, though endpoints MAY choose to treat frames that arrive a significant time after sending END_STREAM as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

WINDOW_UPDATE或RST_STREAM帧可以在收到带有END_STREAM标识的DATA帧或HEADERS帧之后短暂的时期内被接收。直到远程对等端收到并且处理了RST_STREAM或者带有END_STREAM的帧，它可以发送这两类帧。虽然端点可能会选择在发送END_STREAM之后很长时间之后收到帧视为是协议错误类型的连接错误，但是端点必须忽略关闭状态下收到的WINDOW_UPDATE或RST_STREAM帧。

PRIORITY frames can be sent on closed streams to prioritize streams that are dependent on the closed stream. Endpoints SHOULD process PRIORITY frames, though they can be ignored if the stream has been removed from the dependency tree (see Section 5.3.4).

PRIORITY帧可以在已关闭的流中发送来实现设置依赖于这个已关闭的流的流的优先级。尽管当流已经从依赖树种被移除之后可以忽略PRIORITY帧，但端点还是应该处理PRIORITY帧。

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM might have already sent -- or enqueued for sending -- frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

当流因为发送RST_STREAM帧而变为关闭状态时，收到RST_STREAM帧的对等端可能已经往流中发送，或者准备发送无法撤销的帧。所以端点必须忽视在发送RST_STREAM帧时候收到的帧。端点可以选择在一段时间内忽略这些帧，然后超过这个时间到达的帧都视为错误。

Flow-controlled frames (i.e., DATA) received after sending RST_STREAM are counted toward the connection flow-control window. Even though these frames might be ignored, because they are sent before the sender receives the RST_STREAM, the sender will consider the frames to count against the flow-control window.

在发送RST_STREAM之后收到的流量控制帧统计入连接的流量控制窗口。虽然这些帧可以被忽略，但是它们是在发送者收到RST_STREAM之前发送的帧，所以发送者会认为这些帧仍然应该计入流量控制窗口。

An endpoint might receive a PUSH_PROMISE frame after it sends RST_STREAM. PUSH_PROMISE causes a stream to become "reserved" even if the associated stream has been reset. Therefore, a RST_STREAM is needed to close an unwanted promised stream. In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. Note that PRIORITY can be sent and received in any stream state. Frames of unknown types are ignored.

端点可能会在发送RST_STREAM之后收到一个PUSH_PROMISE帧。即使在流已经被重置的情况下，PUSH_PROMISE帧也会引起流状态变为保留状态。因此，需要一个RST_STREAM帧来关闭一个不希望被承诺的流。由于这个文档缺乏更明确的指导，实现者应该收到在相应状态下没有明确允许的帧时都处理成连接错误。PRIORITY可以在任何状态下发送或者接收。忽略未知类型的帧。

An example of the state transitions for an HTTP request/response exchange can be found in Section 8.1. An example of the state transitions for server push can be found in Sections 8.2.1 and 8.2.2.

Streams are identified with an unsigned 31-bit integer. Streams initiated by a client MUST use odd-numbered stream identifiers; those initiated by the server MUST use even-numbered stream identifiers. A stream identifier of zero (0x0) is used for connection control messages; the stream identifier of zero cannot be used to establish a new stream.

流依靠一个32位无符号整数来作为ID标识。客户端初始化的流必须是奇数的ID，服务器初始化的流必须是偶数的ID。ID为0的流是用来传递连接控制消息的，0不能用来建立一个新的流。

HTTP/1.1 requests that are upgraded to HTTP/2 (see Section 3.2) are responded to with a stream identifier of one (0x1). After the upgrade completes, stream 0x1 is "half-closed (local)" to the client. Therefore, stream 0x1 cannot be selected as a new stream identifier by a client that upgrades from HTTP/1.1.

从HTTP/1.1升级到HTTP/2的请求使用ID为1的流来响应。在升级完成后，ID为1的流对于客户端来时变为“本地半关闭”状态。所以从协议HTTP/1.1升级的客户端不能使用1作为流ID。

The identifier of a newly established stream MUST be numerically greater than all streams that the initiating endpoint has opened or reserved. This governs streams that are opened using a HEADERS frame and streams that are reserved using PUSH_PROMISE. An endpoint that receives an unexpected stream identifier MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

新建立的流的ID必须大于所有打开或预留的流ID。使用HEADERS帧而在打开状态的流和因为PUSH_PROMISE而处于预留状态的流都符合这个规则。如果端点收到不符合条件的流ID，必须返回连接错误。

The first use of a new stream identifier implicitly closes all streams in the "idle" state that might have been initiated by that peer with a lower-valued stream identifier. For example, if a client sends a HEADERS frame on stream 7 without ever sending a frame on stream 5, then stream 5 transitions to the "closed" state when the first frame for stream 7 is sent or received.

第一次使用一个新的流ID会隐式的关闭所有小于这个流ID的所有空闲状态的流。例如，如果ID为5的流没有发送或接收任何帧的情况下，客户端在ID为7的流上发送一个HEADERS帧，那ID为5的流就会变为关闭状态。

Stream identifiers cannot be reused. Long-lived connections can result in an endpoint exhausting the available range of stream identifiers. A client that is unable to establish a new stream identifier can establish a new connection for new streams. A server that is unable to establish a new stream identifier can send a GOAWAY frame so that the client is forced to open a new connection for new streams.

流ID不能重复使用。长连接可能导致端点用完所有可用的流ID。当客户端无法建立一个新的流ID时可以通过建立一个新的连接来创建新的流。如果服务器无法建立新的流ID时可以发送GOAWAY帧给客户端来强制客户端开一个新的连接。

A peer can limit the number of concurrently active streams using the SETTINGS_MAX_CONCURRENT_STREAMS parameter (see Section 6.5.2) within a SETTINGS frame. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate.

端点可以通过使用SETTINGS帧的SETTINGS_MAX_CONCURRENT_STREAMS参数来限制活跃并发流的数量。最大并发流设置只对收到SETTINGS帧的端点生效，每个端点都拥有自己特定的并发数量限制。也就是说，客户端指定的最大并发数量限制了服务器可以初始化的最大流数量，而服务器指定的最大并发流数量则是对客户端生效。

Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the SETTINGS_MAX_CONCURRENT_STREAMS setting. Streams in either of the "reserved" states do not count toward the stream limit.

打开状态和本地半关闭、远程半关闭状态的流都计入端点的最大允许打开的流数量中。这三个状态的流都需要计入SETTINGS_MAX_CONCURRENT_STREAMS参数的限额中。预留状态的流不计入限额。

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a HEADERS frame that causes its advertised concurrent stream limit to be exceeded MUST treat this as a stream error (Section 5.4.2) of type PROTOCOL_ERROR or REFUSED_STREAM. The choice of error code determines whether the endpoint wishes to enable automatic retry (see Section 8.1.4) for details).

端点必须不能超过对方设置的并发限制。当端点收到一个HEADERS帧导致它超过了并发流限制，则必须处理成PROTOCOL_ERROR或REFUSED_STREAM类型的流程错误。错误类型的选择取决于端点是否希望允许自动重试。

An endpoint that wishes to reduce the value of SETTINGS_MAX_CONCURRENT_STREAMS to a value that is below the current number of open streams can either close streams that exceed the new value or allow streams to complete.

当一个端点希望将并发数量减少到当前已打开的流数量之下时，它可以选择关闭流来执行这个要求或者选择允许流完成它的剩余工作。

Using streams for multiplexing introduces contention over use of the TCP connection, resulting in blocked streams. A flow-control scheme ensures that streams on the same connection do not destructively interfere with each other. Flow control is used for both individual streams and for the connection as a whole.

HTTP/2 provides for flow control through use of the WINDOW_UPDATE frame (Section 6.9).

使用流的多路复用会引发对TCP连接的过度使用，最终导致流被阻塞。流量控制机制保障同一个连接的流之间不会相互产生破坏性的影响。流量控制既用于单独的流也用户整个连接。

HTTP/2使用WINDOW_UPDATE帧来提供流量控制服务。

HTTP/2 stream flow control aims to allow a variety of flow-control algorithms to be used without requiring protocol changes. Flow control in HTTP/2 has the following characteristics:

HTTP/2的流量控制目的在于实现在不改变协议的前提下允许使用多种流量控制算法。HTTP/2的流量控制拥有以下的特征。

1. Flow control is specific to a connection. Both types of flow control are between the endpoints of a single hop and not over the entire end-to-end path.
2. Flow control is based on WINDOW_UPDATE frames. Receivers advertise how many octets they are prepared to receive on a stream and for the entire connection. This is a credit-based scheme.
3. Flow control is directional with overall control provided by the receiver. A receiver MAY choose to set any window size that it desires for each stream and for the entire connection. A sender MUST respect flow-control limits imposed by a receiver. Clients, servers, and intermediaries all independently advertise their flow-control window as a receiver and abide by the flow-control limits set by their peer when sending.
4. The initial value for the flow-control window is 65,535 octets for both new streams and the overall connection.
5. The frame type determines whether flow control applies to a frame. Of the frames specified in this document, only DATA frames are subject to flow control; all other frame types do not consume space in the advertised flow-control window. This ensures that important control frames are not blocked by flow control.
6. Flow control cannot be disabled.
7. HTTP/2 defines only the format and semantics of the WINDOW_UPDATE frame (Section 6.9). This document does not stipulate how a receiver decides when to send this frame or the value that it sends, nor does it specify how a sender chooses to send packets. Implementations are able to select any algorithm that suits their needs.

Implementations are also responsible for managing how requests and responses are sent based on priority, choosing how to avoid head-of-line blocking for requests, and managing the creation of new streams. Algorithm choices for these could interact with any flow-control algorithm.

1. 流量控制是特定于连接的。两种类型的流量控制都是基于网路中相邻的两个端点间，而不是整个端到端的路径。
2. 流量控制基于WINDOW_UPDATE帧。接收者广播它在一个流和整个连接中准备接收多少字节。这是一个基于信用的方案。
3. 流量控制是由接收器提供的有方向性的全面控制。发送方必须尊重由接收方设置的流量控制范围。客户端、服务器和路由器等都独立的广播他们作为接收者的流量控制窗口与此同时准守他们的接收者所设定的流量控制窗口限制。
4. 新流和新连接的初始的流量控制窗口大小都是65535字节。
5. 帧类型决定了流量控制是否适用于这个帧。在这个文档中指定的帧，只有数据帧受到流

量控制的约束；其他类型的帧都不会消耗流量控制窗口的空间。这保证了重要的控制帧不会被流量控制阻塞。

6. 流量控制不能被禁止。
7. HTTP/2仅仅定义了WINDOW_UPDATE帧的格式和含义。文档并没有规定接收者如何决定何时到那个这个帧以及这个帧发送的值，也没有规定发送者选择怎样发送包。实现者可以选择合适自己的任何算法。

负责管理如何基于优先级管理请求和响应发送、选择如何避免请求的排头阻塞以及管理新流的创建。这些算法的选择需要能够和所有流量控制算法交互。接收者可以为每个流和整个连接设置它所期望的任何窗口大小。

Flow control is defined to protect endpoints that are operating under resource constraints. For example, a proxy needs to share memory between many connections and also might have a slow upstream connection and a fast downstream one. Flow-control addresses cases where the receiver is unable to process data on one stream yet wants to continue to process other streams in the same connection.

流量控制是设计用来保护在有限资源限制的情况下运作的端点。例如，一个代理需要在很多连接之间共享内存，它可能有一个非常慢的上游和一个非常快的下游。流量控制解决如下情况下的问题：接收者没法处理某个流的数据但是它还想继续处理同一个连接中的其他流。

Deployments that do not require this capability can advertise a flow-control window of the maximum size (2^31-1) and can maintain this window by sending a WINDOW_UPDATE frame when any data is received. This effectively disables flow control for that receiver. Conversely, a sender is always subject to the flow-control window advertised by the receiver.

不需要使用流量控制的应用可以将控制窗口大小设置到最大，然后在收到数据后发送WINDOW_UPDATE帧来维持窗口大小。这样可以有效的禁止流量控制。相反的，发送者永远受到接收者流量控制窗口的控制。

Deployments with constrained resources (for example, memory) can employ flow control to limit the amount of memory a peer can consume. Note, however, that this can lead to suboptimal use of available network resources if flow control is enabled without knowledge of the bandwidth-delay product (see [RFC7323]).

资源（内存）受限的情况下可以使用流量控制来限制一个连接使用的内存量。然而需要注意，如果在不知道时延带宽积时使用流量控制将导致无法最大化利用网络资源。

Even with full awareness of the current bandwidth-delay product, implementation of flow control can be difficult. When using flow control, the receiver MUST read from the TCP receive buffer in a timely fashion. Failure to do so could lead to a deadlock when critical frames, such as WINDOW_UPDATE, are not read and acted upon.

即使对当前的时延带宽积非常清楚，实现流量控制也是非常困难的。当使用流量控制时，接收者必须即使的从TCP接收缓存中读取数据。如果一些关键的帧例如WINDOW_UPDATE帧没能及时读取和应用将会导致死锁。

A client can assign a priority for a new stream by including prioritization information in the HEADERS frame (Section 6.2) that opens the stream. At any other time, the PRIORITY frame (Section 6.3) can be used to change the priority of a stream.

客户端可以通过在打开流的HEADERS帧包含优先级信息来赋予新流优先级。在任意时刻，都可以通过PRIORITY帧来改变流的优先级。

The purpose of prioritization is to allow an endpoint to express how it would prefer its peer to allocate resources when managing concurrent streams. Most importantly, priority can be used to select streams for transmitting frames when there is limited capacity for sending.

优先级的目的是允许端点表达它期望对方在管理并发的流时如何分配资源。更重要的是，优先级可以用来在发送容量受限的情况下决定哪些流可以发送数据。

Streams can be prioritized by marking them as dependent on the completion of other streams (Section 5.3.1). Each dependency is assigned a relative weight, a number that is used to determine the relative proportion of available resources that are assigned to streams dependent on the same stream.

流可以通过标记为依赖其他流的完成来设置优先级。每个依赖会分配一个相对的权重，这个权重决定依赖同一个流的所有流之间分配可用资源的比例。

Explicitly setting the priority for a stream is input to a prioritization process. It does not guarantee any particular processing or transmission order for the stream relative to any other stream. An endpoint cannot force a peer to process concurrent streams in a particular order using priority. Expressing priority is therefore only a suggestion.

给流明确的设置优先级会被输入到优先级处理过程中。这个过程不保证特殊的处理或者是改变流之间的相对处理顺序。端点不能通过优先级强迫对等端按指定的顺序处理并发的流。因此，指明优先级只是一个推荐。

Prioritization information can be omitted from messages. Defaults are used prior to any explicit values being provided (Section 5.3.5).

优先级信息可以从信息中删除。默认的优先级先于任何明确的优先级值。

Each stream can be given an explicit dependency on another stream. Including a dependency expresses a preference to allocate resources to the identified stream rather than to the dependent stream.

每个流都可以明确地依赖另一个流。包含一个依赖表示倾向于分配更多的资源给被依赖的流而不是依赖其他流的流。

A stream that is not dependent on any other stream is given a stream dependency of 0x0. In other words, the non-existent stream 0 forms the root of the tree.

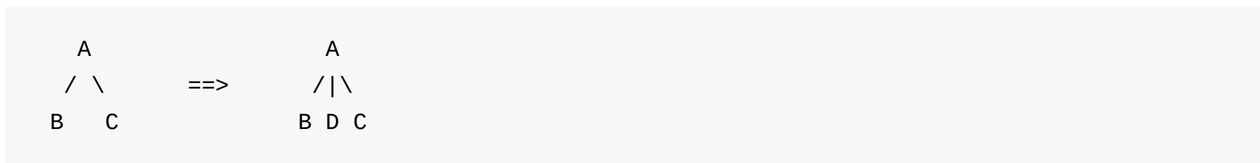
一个没有依赖任何流的流会被赋予0作为依赖。换句话说，不存在的流0形成了依赖树的更节点。

A stream that depends on another stream is a dependent stream. The stream upon which a stream is dependent is a parent stream. A dependency on a stream that is not currently in the tree -- such as a stream in the "idle" state -- results in that stream being given a default priority (Section 5.3.5).

遗留依赖于其他流的流叫做依赖流。被依赖的流叫做父流。依赖当前不在依赖树中的流会导致这个被依赖的流被赋予一个默认优先级（例如依赖一个空闲状态的流）。

When assigning a dependency on another stream, the stream is added as a new dependency of the parent stream. Dependent streams that share the same parent are not ordered with respect to each other. For example, if streams B and C are dependent on stream A, and if stream D is created with a dependency on stream A, this results in a dependency order of A followed by B, C, and D in any order.

当分配一个依赖时，这个流被加到父流的依赖树中。拥有共同父节点的依赖流是没有固定顺序的。例如，B和C依赖于A，当D也创建一个依赖依赖于A后，B、C和D的顺序完全是随机的，可以是任何顺序。

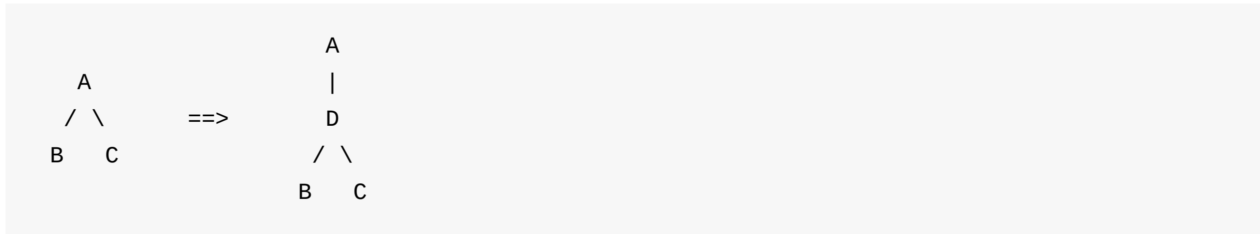


```
  A                 A
 / \      ==>      /|\
B   C             B D C
```

Figure 3: Example of Default Dependency Creation

An exclusive flag allows for the insertion of a new level of dependencies. The exclusive flag causes the stream to become the sole dependency of its parent stream, causing other dependencies to become dependent on the exclusive stream. In the previous example, if stream D is created with an exclusive dependency on stream A, this results in D becoming the dependency parent of B and C.

排外标志允许插入一个新级别的依赖。排外标志导致这个流变成父流唯一的依赖流，而其他的依赖流则会成为它的子流。在上面的例子中，如果D创建一个排外依赖依赖于A，则D将成为B和C的父流。



```
                     A
     A               |
    / \    ==>       D
   B   C            / \
                   B   C
```

Figure 4: Example of Exclusive Dependency Creation

Inside the dependency tree, a dependent stream SHOULD only be allocated resources if either all of the streams that it depends on (the chain of parent streams up to 0x0) are closed or it is not possible to make progress on them.

在依赖树里面，依赖流仅可以在所有它所有的父流都关闭了或者不能操作的时候才能分配到资源。

A stream cannot depend on itself. An endpoint MUST treat this as a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

流不能依赖自己。端点必须将依赖自己视为流错误。

All dependent streams are allocated an integer weight between 1 and 256 (inclusive).

所有的依赖流都会分配一个1到256之间的整数作为权重。

Streams with the same parent SHOULD be allocated resources proportionally based on their weight. Thus, if stream B depends on stream A with weight 4, stream C depends on stream A with weight 12, and no progress can be made on stream A, stream B ideally receives one-third of the resources allocated to stream C.

拥有相同父流的流依据它们的权重比例来分配资源。一次，如果B和C都依赖于A，B的权重为4，C的权重为12，当流A不能操作时，B获得的资源是C的1/3。

Stream priorities are changed using the PRIORITY frame. Setting a dependency causes a stream to become dependent on the identified parent stream.

使用PRIORITY帧可以改变流的优先级。给一个流设置依赖使得这个流变成父类的依赖流。

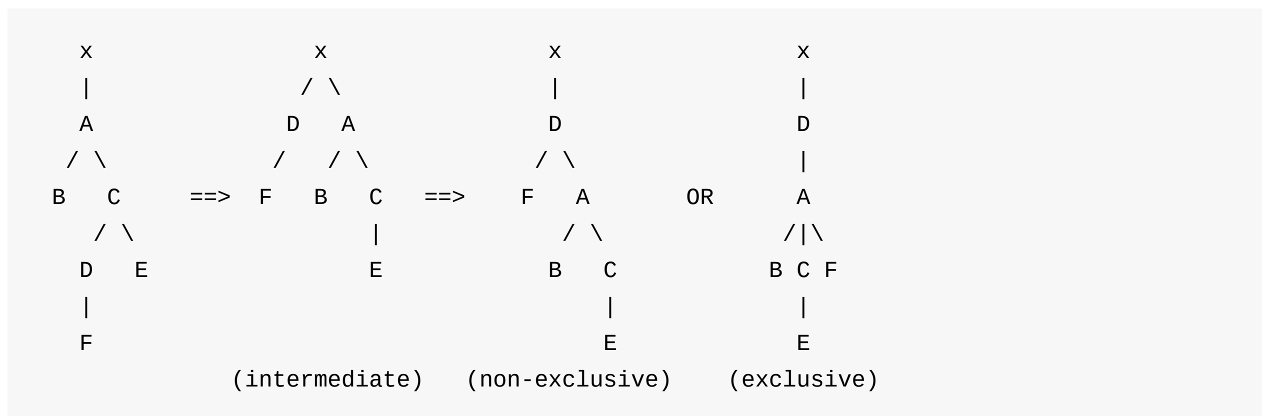
Dependent streams move with their parent stream if the parent is reprioritized. Setting a dependency with the exclusive flag for a reprioritized stream causes all the dependencies of the new parent stream to become dependent on the reprioritized stream.

如果父流重设了优先级，那依赖流的优先级也随之改变。给重置优先级的流S设置依赖的时候带上排外标志会导致这个父流R的所有之前的依赖都变成S的依赖流。

If a stream is made dependent on one of its own dependencies, the formerly dependent stream is first moved to be dependent on the reprioritized stream's previous parent. The moved dependency retains its weight.

如果将一个流B设置依赖它本身的依赖流C（假设C依赖于B，B依赖于A），那么首先将依赖流C设置为B的父流C。被移动的依赖仍然保持之前的权重。

For example, consider an original dependency tree where B and C depend on A, D and E depend on C, and F depends on D. If A is made dependent on D, then D takes the place of A. All other dependency relationships stay the same, except for F, which becomes dependent on A if the reprioritization is exclusive.



```
    x                x                x                  x
    |               / \               |                  |
    A              D   A              D                  D
   / \            /   / \            / \                 |
  B   C     ==>  F   B   C   ==>    F   A       OR       A
     / \                 |             / \              /|\
    D   E                E            B   C            B C F
    |                                     |              |
    F                                     E              E
             (intermediate)   (non-exclusive)    (exclusive)
```

Figure 5: Example of Dependency Reordering

例如，上图中，B、C依赖于A，D、E依赖于C，F依赖于D。如果需要让A依赖于D，那么先将D替代A的位置，使得D依赖于A的父流x，然后再让让A依赖于D，形成上图第3个依赖树的形状。如果设置了排外标志，那意味着只允许A依赖于D，之前依赖于D的流都需要下移依赖于A，形成上图第4个依赖树。

When a stream is removed from the dependency tree, its dependencies can be moved to become dependent on the parent of the closed stream. The weights of new dependencies are recalculated by distributing the weight of the dependency of the closed stream proportionally based on the weights of its dependencies.

Streams that are removed from the dependency tree cause some prioritization information to be lost. Resources are shared between streams with the same parent stream, which means that if a stream in that set closes or becomes blocked, any spare capacity allocated to a stream is distributed to the immediate neighbors of the stream. However, if the common dependency is removed from the tree, those streams share resources with streams at the next highest level.

For example, assume streams A and B share a parent, and streams C and D both depend on stream A. Prior to the removal of stream A, if streams A and D are unable to proceed, then stream C receives all the resources dedicated to stream A. If stream A is removed from the tree, the weight of stream A is divided between streams C and D. If stream D is still unable to proceed, this results in stream C receiving a reduced proportion of resources. For equal starting weights, C receives one third, rather than one half, of available resources.

It is possible for a stream to become closed while prioritization information that creates a dependency on that stream is in transit. If a stream identified in a dependency has no associated priority information, then the dependent stream is instead assigned a default priority (Section 5.3.5). This potentially creates suboptimal prioritization, since the stream could be given a priority that is different from what is intended.

To avoid these problems, an endpoint SHOULD retain stream prioritization state for a period after streams become closed. The longer state is retained, the lower the chance that streams are assigned incorrect or default priority values.

Similarly, streams that are in the "idle" state can be assigned priority or become a parent of other streams. This allows for the creation of a grouping node in the dependency tree, which enables more flexible expressions of priority. Idle streams begin with a default priority (Section 5.3.5).

The retention of priority information for streams that are not counted toward the limit set by SETTINGS_MAX_CONCURRENT_STREAMS could create a large state burden for an endpoint. Therefore, the amount of prioritization state that is retained MAY be limited.

The amount of additional state an endpoint maintains for prioritization could be dependent on load; under high load, prioritization state can be discarded to limit resource commitments. In extreme cases, an endpoint could even discard prioritization state for active or reserved streams. If a limit is applied, endpoints SHOULD maintain state for at least as many streams

as allowed by their setting for SETTINGS_MAX_CONCURRENT_STREAMS. Implementations SHOULD also attempt to retain state for streams that are in active use in the priority tree.

If it has retained enough state to do so, an endpoint receiving a PRIORITY frame that changes the priority of a closed stream SHOULD alter the dependencies of the streams that depend on it.

All streams are initially assigned a non-exclusive dependency on stream 0x0. Pushed streams (Section 8.2) initially depend on their associated stream. In both cases, streams are assigned a default weight of 16.

HTTP/2 framing permits two classes of error:

- An error condition that renders the entire connection unusable is a connection error.
- An error in an individual stream is a stream error.

A list of error codes is included in Section 7.

A connection error is any error that prevents further processing of the frame layer or corrupts any connection state.

An endpoint that encounters a connection error SHOULD first send a GOAWAY frame (Section 6.8) with the stream identifier of the last stream that it successfully received from its peer. The GOAWAY frame includes an error code that indicates why the connection is terminating. After sending the GOAWAY frame for an error condition, the endpoint MUST close the TCP connection.

It is possible that the GOAWAY will not be reliably received by the receiving endpoint ([RFC7230], Section 6.6 describes how an immediate connection close can result in data loss). In the event of a connection error, GOAWAY only provides a best-effort attempt to communicate with the peer about why the connection is being terminated.

An endpoint can end a connection at any time. In particular, an endpoint MAY choose to treat a stream error as a connection error. Endpoints SHOULD send a GOAWAY frame when ending a connection, providing that circumstances permit it.

A stream error is an error related to a specific stream that does not affect processing of other streams.

An endpoint that detects a stream error sends a RST_STREAM frame (Section 6.4) that contains the stream identifier of the stream where the error occurred. The RST_STREAM frame includes an error code that indicates the type of error.

A RST_STREAM is the last frame that an endpoint can send on a stream. The peer that sends the RST_STREAM frame MUST be prepared to receive any frames that were sent or enqueued for sending by the remote peer. These frames can be ignored, except where they modify connection state (such as the state maintained for header compression (Section 4.3) or flow control).

Normally, an endpoint SHOULD NOT send more than one RST_STREAM frame for any stream. However, an endpoint MAY send additional RST_STREAM frames if it receives frames on a closed stream after more than a round-trip time. This behavior is permitted to deal with misbehaving implementations.

To avoid looping, an endpoint MUST NOT send a RST_STREAM in response to a RST_STREAM frame.

If the TCP connection is closed or reset while streams remain in "open" or "half-closed" state, then the affected streams cannot be automatically retried (see Section 8.1.4 for details).

HTTP/2 permits extension of the protocol. Within the limitations described in this section, protocol extensions can be used to provide additional services or alter any aspect of the protocol. Extensions are effective only within the scope of a single HTTP/2 connection.

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or header fields.

Extensions are permitted to use new frame types (Section 4.1), new settings (Section 6.5.2), or new error codes (Section 7). Registries are established for managing these extension points: frame types (Section 11.2), settings (Section 11.3), and error codes (Section 11.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, extension frames that appear in the middle of a header block (Section 4.3) are not permitted; these MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Extensions that could change the semantics of existing protocol components MUST be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. In this case, it could also be necessary to coordinate when the revised layout comes into effect. Note that treating any frames other than DATA frames as flow controlled is such a change in semantics and can only be done through negotiation.

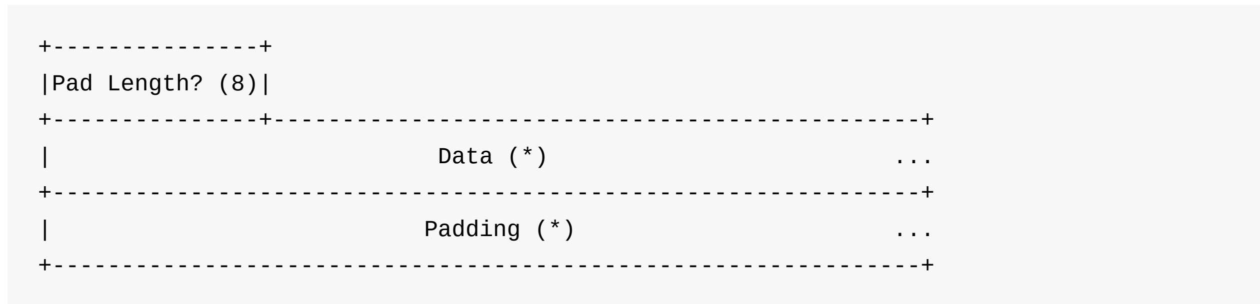
This document doesn't mandate a specific method for negotiating the use of an extension but notes that a setting (Section 6.5.2) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the initial value MUST be defined in such a fashion that the extension is initially disabled.

This specification defines a number of frame types, each identified by a unique 8-bit type code. Each frame type serves a distinct purpose in the establishment and management either of the connection as a whole or of individual streams.

The transmission of specific frame types can alter the state of a connection. If endpoints fail to maintain a synchronized view of the connection state, successful communication within the connection will no longer be possible. Therefore, it is important that endpoints have a shared comprehension of how the state is affected by the use any given frame.

DATA frames (type=0x0) convey arbitrary, variable-length sequences of octets associated with a stream. One or more DATA frames are used, for instance, to carry HTTP request or response payloads.

DATA frames MAY also contain padding. Padding can be added to DATA frames to obscure the size of messages. Padding is a security feature; see Section 10.7.



```
+---------------+
|Pad Length? (8)|
+---------------+-----------------------------------------------+
|                            Data (*)                         ...
+---------------------------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 6: DATA Frame Payload

The DATA frame contains the following fields:

Pad Length: An 8-bit field containing the length of the frame padding in units of octets. This field is conditional (as signified by a "?" in the diagram) and is only present if the PADDED flag is set.

Data: Application data. The amount of data is the remainder of the frame payload after subtracting the length of the other fields that are present.

Padding: Padding octets that contain no application semantic value. Padding octets MUST be set to zero when sending. A receiver is not obligated to verify padding but MAY treat non-zero padding as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. The DATA frame defines the following flags:

END_STREAM (0x1): When set, bit 0 indicates that this frame is the last that the endpoint will send for the identified stream. Setting this flag causes the stream to enter one of the "half-closed" states or the "closed" state (Section 5.1).

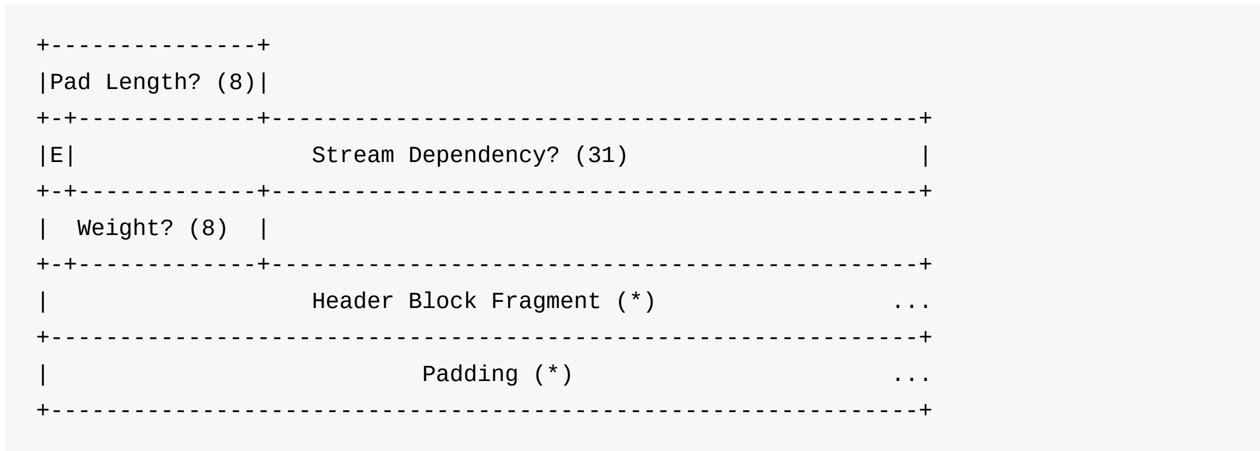
PADDED (0x8): When set, bit 3 indicates that the Pad Length field and any padding that it describes are present. DATA frames MUST be associated with a stream. If a DATA frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

DATA frames are subject to flow control and can only be sent when a stream is in the "open" or "half-closed (remote)" state. The entire DATA frame payload is included in flow control, including the Pad Length and Padding fields if present. If a DATA frame is received whose stream is not in "open" or "half-closed (local)" state, the recipient MUST respond with a stream error (Section 5.4.2) of type STREAM_CLOSED.

The total number of padding octets is determined by the value of the Pad Length field. If the length of the padding is the length of the frame payload or greater, the recipient MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

- Note: A frame can be increased in size by one octet by including a Pad Length field with a value of zero.

The HEADERS frame (type=0x1) is used to open a stream (Section 5.1), and additionally carries a header block fragment. HEADERS frames can be sent on a stream in the "idle", "reserved (local)", "open", or "half-closed (remote)" state.



```
+---------------+
|Pad Length? (8)|
+-+-------------+-----------------------------------------------+
|E|                 Stream Dependency? (31)                     |
+-+-------------+-----------------------------------------------+
|  Weight? (8)  |
+-+-------------+-----------------------------------------------+
|                   Header Block Fragment (*)                 ...
+---------------------------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 7: HEADERS Frame Payload

The HEADERS frame payload has the following fields:

Pad Length: An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.

E: A single-bit flag indicating that the stream dependency is exclusive (see Section 5.3). This field is only present if the PRIORITY flag is set.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on (see Section 5.3). This field is only present if the PRIORITY flag is set.

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see Section 5.3). Add one to the value to obtain a weight between 1 and 256. This field is only present if the PRIORITY flag is set.

Header Block Fragment: A header block fragment (Section 4.3).

Padding: Padding octets. The HEADERS frame defines the following flags:

END_STREAM (0x1): When set, bit 0 indicates that the header block (Section 4.3) is the last that the endpoint will send for the identified stream.

A HEADERS frame carries the END_STREAM flag that signals the end of a stream. However, a HEADERS frame with the END_STREAM flag set can be followed by CONTINUATION frames on the same stream. Logically, the CONTINUATION frames are part of the HEADERS frame.

END_HEADERS (0x4): When set, bit 2 indicates that this frame contains an entire header block (Section 4.3) and is not followed by any CONTINUATION frames.

A HEADERS frame without the END_HEADERS flag set MUST be followed by a CONTINUATION frame for the same stream. A receiver MUST treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

PADDED (0x8): When set, bit 3 indicates that the Pad Length field and any padding that it describes are present.

PRIORITY (0x20): When set, bit 5 indicates that the Exclusive Flag (E), Stream Dependency, and Weight fields are present; see Section 5.3. The payload of a HEADERS frame contains a header block fragment (Section 4.3). A header block that does not fit within a HEADERS frame is continued in a CONTINUATION frame (Section 6.10).

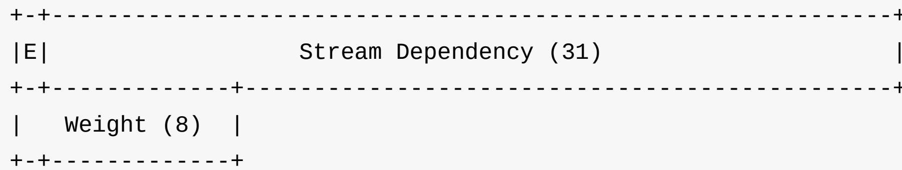
HEADERS frames MUST be associated with a stream. If a HEADERS frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The HEADERS frame changes the connection state as described in Section 4.3.

The HEADERS frame can include padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1). Padding that exceeds the size remaining for the header block fragment MUST be treated as a PROTOCOL_ERROR.

Prioritization information in a HEADERS frame is logically equivalent to a separate PRIORITY frame, but inclusion in HEADERS avoids the potential for churn in stream prioritization when new streams are created. Prioritization fields in HEADERS frames subsequent to the first on a stream reprioritize the stream (Section 5.3.3).

The PRIORITY frame (type=0x2) specifies the sender-advised priority of a stream (Section 5.3). It can be sent in any stream state, including idle or closed streams.



```
+-+-------------------------------------------------------------+
|E|                 Stream Dependency (31)                      |
+-+-------------+-----------------------------------------------+
|   Weight (8)  |
+-+-------------+
```

Figure 8: PRIORITY Frame Payload

The payload of a PRIORITY frame contains the following fields:

E: A single-bit flag indicating that the stream dependency is exclusive (see Section 5.3).

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on (see Section 5.3).

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see Section 5.3). Add one to the value to obtain a weight between 1 and 256. The PRIORITY frame does not define any flags.

The PRIORITY frame always identifies a stream. If a PRIORITY frame is received with a stream identifier of 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The PRIORITY frame can be sent on a stream in any state, though it cannot be sent between consecutive frames that comprise a single header block (Section 4.3). Note that this frame could arrive after processing or frame sending has completed, which would cause it to have no effect on the identified stream. For a stream that is in the "half-closed (remote)" or "closed" state, this frame can only affect processing of the identified stream and its dependent streams; it does not affect frame transmission on that stream.

The PRIORITY frame can be sent for a stream in the "idle" or "closed" state. This allows for the reprioritization of a group of dependent streams by altering the priority of an unused or closed parent stream.

A PRIORITY frame with a length other than 5 octets MUST be treated as a stream error (Section 5.4.2) of type FRAME_SIZE_ERROR.

The RST_STREAM frame (type=0x3) allows for immediate termination of a stream. RST_STREAM is sent to request cancellation of a stream or to indicate that an error condition has occurred.



```
+---------------------------------------------------------------+
|                        Error Code (32)                        |
+---------------------------------------------------------------+
```

Figure 9: RST_STREAM Frame Payload

The RST_STREAM frame contains a single unsigned, 32-bit integer identifying the error code (Section 7). The error code indicates why the stream is being terminated.

The RST_STREAM frame does not define any flags.

The RST_STREAM frame fully terminates the referenced stream and causes it to enter the "closed" state. After receiving a RST_STREAM on a stream, the receiver MUST NOT send additional frames for that stream, with the exception of PRIORITY. However, after sending the RST_STREAM, the sending endpoint MUST be prepared to receive and process additional frames sent on the stream that might have been sent by the peer prior to the arrival of the RST_STREAM.

RST_STREAM frames MUST be associated with a stream. If a RST_STREAM frame is received with a stream identifier of 0x0, the recipient MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

RST_STREAM frames MUST NOT be sent for a stream in the "idle" state. If a RST_STREAM frame identifying an idle stream is received, the recipient MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

A RST_STREAM frame with a length other than 4 octets MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. The SETTINGS frame is also used to acknowledge the receipt of those parameters. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which are used by the receiving peer. Different values for the same parameter can be advertised by each peer. For example, a client might set a high initial flow-control window, whereas a server might set a lower value to conserve resources.

A SETTINGS frame MUST be sent by both endpoints at the start of a connection and MAY be sent at any other time by either endpoint over the lifetime of the connection. Implementations MUST support all of the parameters defined by this specification.

Each parameter in a SETTINGS frame replaces any existing value for that parameter. Parameters are processed in the order in which they appear, and a receiver of a SETTINGS frame does not need to maintain any state other than the current value of its parameters. Therefore, the value of a SETTINGS parameter is the last value that is seen by a receiver.

SETTINGS parameters are acknowledged by the receiving peer. To enable this, the SETTINGS frame defines the following flag:

ACK (0x1): When set, bit 0 indicates that this frame acknowledges receipt and application of the peer's SETTINGS frame. When this bit is set, the payload of the SETTINGS frame MUST be empty. Receipt of a SETTINGS frame with the ACK flag set and a length field value other than 0 MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR. For more information, see Section 6.5.3 ("Settings Synchronization"). SETTINGS frames always apply to a connection, never a single stream. The stream identifier for a SETTINGS frame MUST be zero (0x0). If an endpoint receives a SETTINGS frame whose stream identifier field is anything other than 0x0, the endpoint MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

A SETTINGS frame with a length other than a multiple of 6 octets MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and an unsigned 32-bit value.



```
+-------------------------------+
|       Identifier (16)         |
+-------------------------------+-------------------------------+
|                        Value (32)                             |
+---------------------------------------------------------------+
```

Figure 10: Setting Format

The following parameters are defined:

SETTINGS_HEADER_TABLE_SIZE (0x1): Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block (see [COMPRESSION]). The initial value is 4,096 octets.

SETTINGS_ENABLE_PUSH (0x2): This setting can be used to disable server push (Section 8.2). An endpoint MUST NOT send a PUSH_PROMISE frame if it receives this parameter set to a value of 0. An endpoint that has both set this parameter to 0 and had it acknowledged MUST treat the receipt of a PUSH_PROMISE frame as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The initial value is 1, which indicates that server push is permitted. Any value other than 0 or 1 MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

SETTINGS_MAX_CONCURRENT_STREAMS (0x3): Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially, there is no limit to this value. It is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

A value of 0 for SETTINGS_MAX_CONCURRENT_STREAMS SHOULD NOT be treated as special by endpoints. A zero value does prevent the creation of new streams; however, this can also happen for any limit that is exhausted with active streams. Servers SHOULD only set a zero value for short durations; if a server does not wish to accept requests, closing the connection is more appropriate.

SETTINGS_INITIAL_WINDOW_SIZE (0x4): Indicates the sender's initial window size (in octets) for stream-level flow control. The initial value is 2^16-1 (65,535) octets.

This setting affects the window size of all streams (see Section 6.9.2).

Values above the maximum flow-control window size of 2^31-1 MUST be treated as a connection error (Section 5.4.1) of type FLOW_CONTROL_ERROR.

SETTINGS_MAX_FRAME_SIZE (0x5): Indicates the size of the largest frame payload that the sender is willing to receive, in octets.

The initial value is 2^14 (16,384) octets. The value advertised by an endpoint MUST be between this initial value and the maximum allowed frame size (2^24-1 or 16,777,215 octets), inclusive. Values outside this range MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

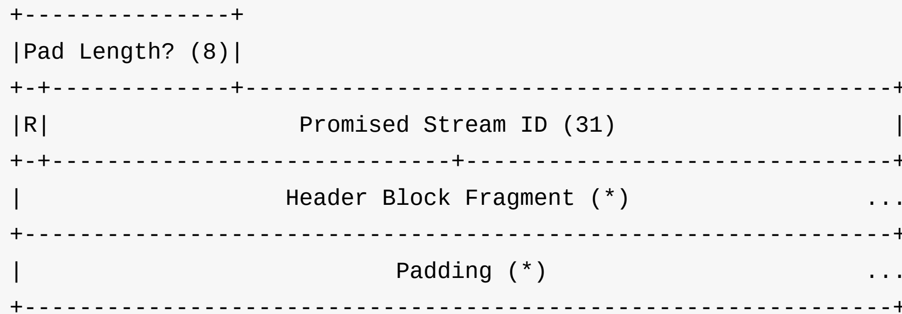
For any given request, a lower limit than what is advertised MAY be enforced. The initial value of this setting is unlimited. An endpoint that receives a SETTINGS frame with any unknown or unsupported identifier MUST ignore that setting.

Most values in SETTINGS benefit from or require an understanding of when the peer has received and applied the changed parameter values. In order to provide such synchronization timepoints, the recipient of a SETTINGS frame in which the ACK flag is not set MUST apply the updated parameters as soon as possible upon receipt.

The values in the SETTINGS frame MUST be processed in the order they appear, with no other frame processing between values. Unsupported parameters MUST be ignored. Once all values have been processed, the recipient MUST immediately emit a SETTINGS frame with the ACK flag set. Upon receiving a SETTINGS frame with the ACK flag set, the sender of the altered parameters can rely on the setting having been applied.

If the sender of a SETTINGS frame does not receive an acknowledgement within a reasonable amount of time, it MAY issue a connection error (Section 5.4.1) of type SETTINGS_TIMEOUT.

The PUSH_PROMISE frame (type=0x5) is used to notify the peer endpoint in advance of streams the sender intends to initiate. The PUSH_PROMISE frame includes the unsigned 31-bit identifier of the stream the endpoint plans to create along with a set of headers that provide additional context for the stream. Section 8.2 contains a thorough description of the use of PUSH_PROMISE frames.



```
+---------------+
|Pad Length? (8)|
+-+-------------+-----------------------------------------------+
|R|                  Promised Stream ID (31)                    |
+-+-----------------------------+-------------------------------+
|                   Header Block Fragment (*)                 ...
+---------------------------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 11: PUSH_PROMISE Payload Format

The PUSH_PROMISE frame payload has the following fields:

Pad Length: An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.

R: A single reserved bit.

Promised Stream ID: An unsigned 31-bit integer that identifies the stream that is reserved by the PUSH_PROMISE. The promised stream identifier MUST be a valid choice for the next stream sent by the sender (see "new stream identifier" in Section 5.1.1).

Header Block Fragment: A header block fragment (Section 4.3) containing request header fields.

Padding: Padding octets. The PUSH_PROMISE frame defines the following flags:

END_HEADERS (0x4): When set, bit 2 indicates that this frame contains an entire header block (Section 4.3) and is not followed by any CONTINUATION frames.

A PUSH_PROMISE frame without the END_HEADERS flag set MUST be followed by a CONTINUATION frame for the same stream. A receiver MUST treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

PADDED (0x8): When set, bit 3 indicates that the Pad Length field and any padding that it describes are present. PUSH_PROMISE frames MUST only be sent on a peer-initiated stream that is in either the "open" or "half-closed (remote)" state. The stream identifier of a

PUSH_PROMISE frame indicates the stream it is associated with. If the stream identifier field specifies the value 0x0, a recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Promised streams are not required to be used in the order they are promised. The PUSH_PROMISE only reserves stream identifiers for later use.

PUSH_PROMISE MUST NOT be sent if the SETTINGS_ENABLE_PUSH setting of the peer endpoint is set to 0. An endpoint that has set this setting and has received acknowledgement MUST treat the receipt of a PUSH_PROMISE frame as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Recipients of PUSH_PROMISE frames can choose to reject promised streams by returning a RST_STREAM referencing the promised stream identifier back to the sender of the PUSH_PROMISE.

A PUSH_PROMISE frame modifies the connection state in two ways. First, the inclusion of a header block (Section 4.3) potentially modifies the state maintained for header compression. Second, PUSH_PROMISE also reserves a stream for later use, causing the promised stream to enter the "reserved" state. A sender MUST NOT send a PUSH_PROMISE on a stream unless that stream is either "open" or "half-closed (remote)"; the sender MUST ensure that the promised stream is a valid choice for a new stream identifier (Section 5.1.1) (that is, the promised stream MUST be in the "idle" state).

Since PUSH_PROMISE reserves a stream, ignoring a PUSH_PROMISE frame causes the stream state to become indeterminate. A receiver MUST treat the receipt of a PUSH_PROMISE on a stream that is neither "open" nor "half-closed (local)" as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. However, an endpoint that has sent RST_STREAM on the associated stream MUST handle PUSH_PROMISE frames that might have been created before the RST_STREAM frame is received and processed.

A receiver MUST treat the receipt of a PUSH_PROMISE that promises an illegal stream identifier (Section 5.1.1) as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. Note that an illegal stream identifier is an identifier for a stream that is not currently in the "idle" state.

The PUSH_PROMISE frame can include padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1).

The PING frame (type=0x6) is a mechanism for measuring a minimal round-trip time from the sender, as well as determining whether an idle connection is still functional. PING frames can be sent from any endpoint.



```
+---------------------------------------------------------------+
|                                                               |
|                      Opaque Data (64)                         |
|                                                               |
+---------------------------------------------------------------+
```

Figure 12: PING Payload Format

In addition to the frame header, PING frames MUST contain 8 octets of opaque data in the payload. A sender can include any value it chooses and use those octets in any fashion.

Receivers of a PING frame that does not include an ACK flag MUST send a PING frame with the ACK flag set in response, with an identical payload. PING responses SHOULD be given higher priority than any other frame.

The PING frame defines the following flags:

ACK (0x1): When set, bit 0 indicates that this PING frame is a PING response. An endpoint MUST set this flag in PING responses. An endpoint MUST NOT respond to PING frames containing this flag. PING frames are not associated with any individual stream. If a PING frame is received with a stream identifier field value other than 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Receipt of a PING frame with a length field value other than 8 MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

The GOAWAY frame (type=0x7) is used to initiate shutdown of a connection or to signal serious error conditions. GOAWAY allows an endpoint to gracefully stop accepting new streams while still finishing processing of previously established streams. This enables administrative actions, like server maintenance.

There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY frame. To deal with this case, the GOAWAY contains the stream identifier of the last peer-initiated stream that was or might be processed on the sending endpoint in this connection. For instance, if the server sends a GOAWAY frame, the identified stream is the highest-numbered stream initiated by the client.

Once sent, the sender will ignore frames sent on streams initiated by the receiver if the stream has an identifier higher than the included last stream identifier. Receivers of a GOAWAY frame MUST NOT open additional streams on the connection, although a new connection can be established for new streams.

If the receiver of the GOAWAY has sent data on streams with a higher stream identifier than what is indicated in the GOAWAY frame, those streams are not or will not be processed. The receiver of the GOAWAY frame can treat the streams as though they had never been created at all, thereby allowing those streams to be retried later on a new connection.

Endpoints SHOULD always send a GOAWAY frame before closing a connection so that the remote peer can know whether a stream has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint might choose to close a connection without sending a GOAWAY for misbehaving peers.

A GOAWAY frame might not immediately precede closing of the connection; a receiver of a GOAWAY that has no more use for the connection SHOULD still send a GOAWAY frame before terminating the connection.

```
+-+-------------------------------------------------------------+
|R|                  Last-Stream-ID (31)                        |
+-+-------------------------------------------------------------+
|                      Error Code (32)                          |
+---------------------------------------------------------------+
|                  Additional Debug Data (*)                    |
+---------------------------------------------------------------+
```

Figure 13: GOAWAY Payload Format

The GOAWAY frame does not define any flags.

The GOAWAY frame applies to the connection, not a specific stream. An endpoint MUST treat a GOAWAY frame with a stream identifier other than 0x0 as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The last stream identifier in the GOAWAY frame contains the highest-numbered stream identifier for which the sender of the GOAWAY frame might have taken some action on or might yet take action on. All streams up to and including the identified stream might have been processed in some way. The last stream identifier can be set to 0 if no streams were processed.

- Note: In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result.

If a connection terminates without a GOAWAY frame, the last stream identifier is effectively the highest possible stream identifier.

On streams with lower- or equal-numbered identifiers that were not closed completely prior to the connection being closed, reattempting requests, transactions, or any protocol activity is not possible, with the exception of idempotent actions like HTTP GET, PUT, or DELETE. Any protocol activity that uses higher-numbered streams can be safely retried using a new connection.

Activity on streams numbered lower or equal to the last stream identifier might still complete successfully. The sender of a GOAWAY frame might gracefully shut down a connection by sending a GOAWAY frame, maintaining the connection in an "open" state until all in-progress streams complete.

An endpoint MAY send multiple GOAWAY frames if circumstances change. For instance, an endpoint that sends GOAWAY with NO_ERROR during graceful shutdown could subsequently encounter a condition that requires immediate termination of the connection. The last stream identifier from the last GOAWAY frame received indicates which streams could have been acted upon. Endpoints MUST NOT increase the value they send in the last stream identifier, since the peers might already have retried unprocessed requests on another connection.

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. This is especially true for intermediaries that might not be serving clients using HTTP/2. A server that is attempting to gracefully shut down a connection SHOULD send an initial GOAWAY frame with the last stream identifier set to 2^31-1 and a NO_ERROR code. This signals to the client that a shutdown is imminent and that initiating further requests is prohibited. After allowing time for any in-flight stream creation (at least one round-trip time), the server can send another GOAWAY frame with an updated last stream identifier. This ensures that a connection can be cleanly shut down without losing requests.

After sending a GOAWAY frame, the sender can discard frames for streams initiated by the receiver with identifiers higher than the identified last stream. However, any frames that alter connection state cannot be completely ignored. For instance, HEADERS, PUSH_PROMISE, and CONTINUATION frames MUST be minimally processed to ensure the state maintained for header compression is consistent (see Section 4.3); similarly, DATA frames MUST be counted toward the connection flow-control window. Failure to process these frames can cause flow control or header compression state to become unsynchronized.

The GOAWAY frame also contains a 32-bit error code (Section 7) that contains the reason for closing the connection.

Endpoints MAY append opaque data to the payload of any GOAWAY frame. Additional debug data is intended for diagnostic purposes only and carries no semantic value. Debug information could contain security- or privacy-sensitive data. Logged or otherwise persistently stored debug data MUST have adequate safeguards to prevent unauthorized access.

The WINDOW_UPDATE frame (type=0x8) is used to implement flow control; see Section 5.2 for an overview.

Flow control operates at two levels: on each individual stream and on the entire connection.

Both types of flow control are hop by hop, that is, only between the two endpoints. Intermediaries do not forward WINDOW_UPDATE frames between dependent connections. However, throttling of data transfer by any receiver can indirectly cause the propagation of flow-control information toward the original sender.

Flow control only applies to frames that are identified as being subject to flow control. Of the frame types defined in this document, this includes only DATA frames. Frames that are exempt from flow control MUST be accepted and processed, unless the receiver is unable to assign resources to handling the frame. A receiver MAY respond with a stream error (Section 5.4.2) or connection error (Section 5.4.1) of type FLOW_CONTROL_ERROR if it is unable to accept a frame.

```
+-+-------------------------------------------------------------+
|R|              Window Size Increment (31)                     |
+-+-------------------------------------------------------------+
```

Figure 14: WINDOW_UPDATE Payload Format

The payload of a WINDOW_UPDATE frame is one reserved bit plus an unsigned 31-bit integer indicating the number of octets that the sender can transmit in addition to the existing flow-control window. The legal range for the increment to the flow-control window is 1 to 2^31-1 (2,147,483,647) octets.

The WINDOW_UPDATE frame does not define any flags.

The WINDOW_UPDATE frame can be specific to a stream or to the entire connection. In the former case, the frame's stream identifier indicates the affected stream; in the latter, the value "0" indicates that the entire connection is the subject of the frame.

A receiver MUST treat the receipt of a WINDOW_UPDATE frame with an flow-control window increment of 0 as a stream error (Section 5.4.2) of type PROTOCOL_ERROR; errors on the connection flow-control window MUST be treated as a connection error (Section 5.4.1).

WINDOW_UPDATE can be sent by a peer that has sent a frame bearing the END_STREAM flag. This means that a receiver could receive a WINDOW_UPDATE frame on a "half-closed (remote)" or "closed" stream. A receiver MUST NOT treat this as an error (see Section 5.1).

A receiver that receives a flow-controlled frame MUST always account for its contribution against the connection flow-control window, unless the receiver treats this as a connection error (Section 5.4.1). This is necessary even if the frame is in error. The sender counts the frame toward the flow-control window, but if the receiver does not, the flow-control window at the sender and receiver can become different.

A WINDOW_UPDATE frame with a length other than 4 octets MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

Flow control in HTTP/2 is implemented using a window kept by each sender on every stream. The flow-control window is a simple integer value that indicates how many octets of data the sender is permitted to transmit; as such, its size is a measure of the buffering capacity of the receiver.

Two flow-control windows are applicable: the stream flow-control window and the connection flow-control window. The sender MUST NOT send a flow-controlled frame with a length that exceeds the space available in either of the flow-control windows advertised by the receiver. Frames with zero length with the END_STREAM flag set (that is, an empty DATA frame) MAY be sent if there is no available space in either flow-control window.

For flow-control calculations, the 9-octet frame header is not counted.

After sending a flow-controlled frame, the sender reduces the space available in both windows by the length of the transmitted frame.

The receiver of a frame sends a WINDOW_UPDATE frame as it consumes data and frees up space in flow-control windows. Separate WINDOW_UPDATE frames are sent for the stream- and connection-level flow-control windows.

A sender that receives a WINDOW_UPDATE frame updates the corresponding window by the amount specified in the frame.

A sender MUST NOT allow a flow-control window to exceed 2^31-1 octets. If a sender receives a WINDOW_UPDATE that causes a flow-control window to exceed this maximum, it MUST terminate either the stream or the connection, as appropriate. For streams, the sender sends a RST_STREAM with an error code of FLOW_CONTROL_ERROR; for the connection, a GOAWAY frame with an error code of FLOW_CONTROL_ERROR is sent.

Flow-controlled frames from the sender and WINDOW_UPDATE frames from the receiver are completely asynchronous with respect to each other. This property allows a receiver to aggressively update the window size kept by the sender to prevent streams from stalling.

When an HTTP/2 connection is first established, new streams are created with an initial flow-control window size of 65,535 octets. The connection flow-control window is also 65,535 octets. Both endpoints can adjust the initial window size for new streams by including a value for SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame that forms part of the connection preface. The connection flow-control window can only be changed using WINDOW_UPDATE frames.

Prior to receiving a SETTINGS frame that sets a value for SETTINGS_INITIAL_WINDOW_SIZE, an endpoint can only use the default initial window size when sending flow-controlled frames. Similarly, the connection flow-control window is set to the default initial window size until a WINDOW_UPDATE frame is received.

In addition to changing the flow-control window for streams that are not yet active, a SETTINGS frame can alter the initial flow-control window size for streams with active flow-control windows (that is, streams in the "open" or "half-closed (remote)" state). When the value of SETTINGS_INITIAL_WINDOW_SIZE changes, a receiver MUST adjust the size of all stream flow-control windows that it maintains by the difference between the new value and the old value.

A change to SETTINGS_INITIAL_WINDOW_SIZE can cause the available space in a flow-control window to become negative. A sender MUST track the negative flow-control window and MUST NOT send new flow-controlled frames until it receives WINDOW_UPDATE frames that cause the flow-control window to become positive.

For example, if the client sends 60 KB immediately on connection establishment and the server sets the initial window size to be 16 KB, the client will recalculate the available flow-control window to be -44 KB on receipt of the SETTINGS frame. The client retains a negative flow-control window until WINDOW_UPDATE frames restore the window to being positive, after which the client can resume sending.

A SETTINGS frame cannot alter the connection flow-control window.

An endpoint MUST treat a change to SETTINGS_INITIAL_WINDOW_SIZE that causes any flow-control window to exceed the maximum size as a connection error (Section 5.4.1) of type FLOW_CONTROL_ERROR.

A receiver that wishes to use a smaller flow-control window than the current size can send a new SETTINGS frame. However, the receiver MUST be prepared to receive data that exceeds this window size, since the sender might send data that exceeds the lower limit prior to processing the SETTINGS frame.

After sending a SETTINGS frame that reduces the initial flow-control window size, a receiver MAY continue to process streams that exceed flow-control limits. Allowing streams to continue does not allow the receiver to immediately reduce the space it reserves for flow-control windows. Progress on these streams can also stall, since WINDOW_UPDATE frames are needed to allow the sender to resume sending. The receiver MAY instead send a RST_STREAM with an error code of FLOW_CONTROL_ERROR for the affected streams.

The CONTINUATION frame (type=0x9) is used to continue a sequence of header block fragments (Section 4.3). Any number of CONTINUATION frames can be sent, as long as the preceding frame is on the same stream and is a HEADERS, PUSH_PROMISE, or CONTINUATION frame without the END_HEADERS flag set.



```
+---------------------------------------------------------------+
|                   Header Block Fragment (*)                 ...
+---------------------------------------------------------------+
```

Figure 15: CONTINUATION Frame Payload

The CONTINUATION frame payload contains a header block fragment (Section 4.3).

The CONTINUATION frame defines the following flag:

END_HEADERS (0x4): When set, bit 2 indicates that this frame ends a header block (Section 4.3).

If the END_HEADERS bit is not set, this frame MUST be followed by another CONTINUATION frame. A receiver MUST treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. The CONTINUATION frame changes the connection state as defined in Section 4.3.

CONTINUATION frames MUST be associated with a stream. If a CONTINUATION frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

A CONTINUATION frame MUST be preceded by a HEADERS, PUSH_PROMISE or CONTINUATION frame without the END_HEADERS flag set. A recipient that observes violation of this rule MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Error codes are 32-bit fields that are used in RST_STREAM and GOAWAY frames to convey the reasons for the stream or connection error.

Error codes share a common code space. Some error codes apply only to either streams or the entire connection and have no defined semantics in the other context.

The following error codes are defined:

NO_ERROR (0x0): The associated condition is not a result of an error. For example, a GOAWAY might include this code to indicate graceful shutdown of a connection.

PROTOCOL_ERROR (0x1): The endpoint detected an unspecific protocol error. This error is for use when a more specific error code is not available.

INTERNAL_ERROR (0x2): The endpoint encountered an unexpected internal error.

FLOW_CONTROL_ERROR (0x3): The endpoint detected that its peer violated the flow-control protocol.

SETTINGS_TIMEOUT (0x4): The endpoint sent a SETTINGS frame but did not receive a response in a timely manner. See Section 6.5.3 ("Settings Synchronization").

STREAM_CLOSED (0x5): The endpoint received a frame after a stream was half-closed.

FRAME_SIZE_ERROR (0x6): The endpoint received a frame with an invalid size.

REFUSED_STREAM (0x7): The endpoint refused the stream prior to performing any application processing (see Section 8.1.4 for details).

CANCEL (0x8): Used by the endpoint to indicate that the stream is no longer needed.

COMPRESSION_ERROR (0x9): The endpoint is unable to maintain the header compression context for the connection.

CONNECT_ERROR (0xa): The connection established in response to a CONNECT request (Section 8.3) was reset or abnormally closed.

ENHANCE_YOUR_CALM (0xb): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

INADEQUATE_SECURITY (0xc): The underlying transport has properties that do not meet minimum security requirements (see Section 9.2).

HTTP_1_1_REQUIRED (0xd): The endpoint requires that HTTP/1.1 be used instead of HTTP/2.

Unknown or unsupported error codes MUST NOT trigger any special behavior. These MAY be treated by an implementation as being equivalent to INTERNAL_ERROR.

HTTP/2 is intended to be as compatible as possible with current uses of HTTP. This means that, from the application perspective, the features of the protocol are largely unchanged. To achieve this, all request and response semantics are preserved, although the syntax of conveying those semantics has changed.

Thus, the specification and requirements of HTTP/1.1 Semantics and Content [RFC7231], Conditional Requests [RFC7232], Range Requests [RFC7233], Caching [RFC7234], and Authentication [RFC7235] are applicable to HTTP/2. Selected portions of HTTP/1.1 Message Syntax and Routing [RFC7230], such as the HTTP and HTTPS URI schemes, are also applicable in HTTP/2, but the expression of those semantics for this protocol are defined in the sections below.

A client sends an HTTP request on a new stream, using a previously unused stream identifier (Section 5.1.1). A server sends an HTTP response on the same stream as the request.

An HTTP message (request or response) consists of:

1. for a response only, zero or more HEADERS frames (each followed by zero or more CONTINUATION frames) containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2),
2. one HEADERS frame (followed by zero or more CONTINUATION frames) containing the message headers (see [RFC7230], Section 3.2),
3. zero or more DATA frames containing the payload body (see [RFC7230], Section 3.3), and
4. optionally, one HEADERS frame, followed by zero or more CONTINUATION frames containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

The last frame in the sequence bears an END_STREAM flag, noting that a HEADERS frame bearing the END_STREAM flag can be followed by CONTINUATION frames that carry any remaining portions of the header block.

Other frames (from any stream) MUST NOT occur between the HEADERS frame and any CONTINUATION frames that might follow.

HTTP/2 uses DATA frames to carry message payloads. The chunked transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used in HTTP/2.

Trailing header fields are carried in a header block that also terminates the stream. Such a header block is a sequence starting with a HEADERS frame, followed by zero or more CONTINUATION frames, where the HEADERS frame bears an END_STREAM flag. Header blocks after the first that do not terminate the stream are not part of an HTTP request or response.

A HEADERS frame (and associated CONTINUATION frames) can only appear at the start or end of a stream. An endpoint that receives a HEADERS frame without the END_STREAM flag set after receiving a final (non-informational) status code MUST treat the corresponding request or response as malformed (Section 8.1.2.6).

An HTTP request/response exchange fully consumes a single stream. A request starts with the HEADERS frame that puts the stream into an "open" state. The request ends with a frame bearing END_STREAM, which causes the stream to become "half-closed (local)" for the client and "half-closed (remote)" for the server. A response starts with a HEADERS frame and ends with a frame bearing END_STREAM, which places the stream in the "closed" state.

An HTTP response is complete after the server sends -- or the client receives -- a frame with the END_STREAM flag set (including any CONTINUATION frames needed to complete a header block). A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by sending a RST_STREAM with an error code of NO_ERROR after sending a complete response (i.e., a frame with the END_STREAM flag). Clients MUST NOT discard responses as a result of receiving such a RST_STREAM, though clients can always discard responses at their discretion for other reasons.

HTTP/2 removes support for the 101 (Switching Protocols) informational status code ([RFC7231], Section 6.2.2).

The semantics of 101 (Switching Protocols) aren't applicable to a multiplexed protocol. Alternative protocols are able to use the same mechanisms that HTTP/2 uses to negotiate their use (see Section 3).

HTTP header fields carry information as a series of key-value pairs. For a listing of registered HTTP headers, see the "Message Header Field" registry maintained at https://www.iana.org/assignments/message-headers.

Just as in HTTP/1.x, header field names are strings of ASCII characters that are compared in a case-insensitive fashion. However, header field names MUST be converted to lowercase prior to their encoding in HTTP/2. A request or response containing uppercase header field names MUST be treated as malformed (Section 8.1.2.6).

While HTTP/1.x used the message start-line (see [RFC7230], Section 3.1) to convey the target URI, the method of the request, and the status code for the response, HTTP/2 uses special pseudo-header fields beginning with ':' character (ASCII 0x3a) for this purpose.

Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailers. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 8.1.2.6).

All pseudo-header fields MUST appear in the header block before regular header fields. Any request or response that contains a pseudo-header field that appears in a header block after a regular header field MUST be treated as malformed (Section 8.1.2.6).

HTTP/2 does not use the Connection header field to indicate connection-specific header fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/2 message containing connection-specific header fields; any message containing connection-specific header fields MUST be treated as malformed (Section 8.1.2.6).

The only exception to this is the TE header field, which MAY be present in an HTTP/2 request; when it is, it MUST NOT contain any value other than "trailers".

This means that an intermediary transforming an HTTP/1.x message to HTTP/2 will need to remove any header fields nominated by the Connection header field, along with the Connection header field itself. Such intermediaries SHOULD also remove other connection-specific header fields, such as Keep-Alive, Proxy-Connection, Transfer-Encoding, and Upgrade, even if they are not nominated by the Connection header field.

- Note: HTTP/2 purposefully does not support upgrade to another protocol. The handshake methods described in Section 3 are believed sufficient to negotiate the use of alternative protocols.

The following pseudo-header fields are defined for HTTP/2 requests:

- The :method pseudo-header field includes the HTTP method ([RFC7231], Section 4).
- The :scheme pseudo-header field includes the scheme portion of the target URI ([RFC3986], Section 3.1).

  :scheme is not restricted to http and https schemed URIs. A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

- The :authority pseudo-header field includes the authority portion of the target URI ([RFC3986], Section 3.2). The authority MUST NOT include the deprecated userinfo subcomponent for http or https schemed URIs.

  To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form (see [RFC7230], Section 5.3). Clients that generate HTTP/2 requests directly SHOULD use the :authority pseudo-header field instead of the Host header field. An intermediary that converts an HTTP/2 request to HTTP/1.1 MUST create a Host header field if one is not present in a request by copying the value of the :authority pseudo-header field.

- The :path pseudo-header field includes the path and query parts of the target URI (the path-absolute production and optionally a '?' character followed by the query production (see Sections 3.3 and 3.4 of [RFC3986]). A request in asterisk form includes the value '*' for the :path pseudo-header field.

  This pseudo-header field MUST NOT be empty for http or https URIs; http or https URIs that do not contain a path component MUST include a value of '/'. The exception to this rule is an OPTIONS request for an http or https URI that does not include a path component; these MUST include a :path pseudo-header field with a value of '*' (see [RFC7230], Section 5.3.4).

All HTTP/2 requests MUST include exactly one valid value for the :method, :scheme, and :path pseudo-header fields, unless it is a CONNECT request (Section 8.3). An HTTP request that omits mandatory pseudo-header fields is malformed (Section 8.1.2.6).

HTTP/2 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

For HTTP/2 responses, a single :status pseudo-header field is defined that carries the HTTP status code field (see [RFC7231], Section 6). This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 8.1.2.6).

HTTP/2 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

The Cookie header field [COOKIE] uses a semi-colon (";") to delimit cookie-pairs (or "crumbs"). This header field doesn't follow the list construction rules in HTTP (see [RFC7230], Section 3.2.2), which prevents cookie-pairs from being separated into different name-value pairs. This can significantly reduce compression efficiency as individual cookie-pairs are updated.

To allow for better compression efficiency, the Cookie header field MAY be split into separate header fields, each with one or more cookie-pairs. If there are multiple Cookie header fields after decompression, these MUST be concatenated into a single octet string using the two-octet delimiter of 0x3B, 0x20 (the ASCII string "; ") before being passed into a non-HTTP/2 context, such as an HTTP/1.1 connection, or a generic HTTP server application.

Therefore, the following two lists of Cookie header fields are semantically equivalent.

```
cookie: a=b; c=d; e=f

cookie: a=b
cookie: c=d
cookie: e=f
```

A malformed request or response is one that is an otherwise valid sequence of HTTP/2 frames but is invalid due to the presence of extraneous frames, prohibited header fields, the absence of mandatory header fields, or the inclusion of uppercase header field names.

A request or response that includes a payload body can include a content-length header field. A request or response is also malformed if the value of a content-length header field does not equal the sum of the DATA frame payload lengths that form the body. A response that is defined to have no payload, as described in [RFC7230], Section 3.3.2, can have a non-zero content-length header field, even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

For malformed requests, a server MAY send an HTTP response prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

In HTTP/1.1, an HTTP client is unable to retry a non-idempotent request when an error occurs because there is no means to determine the nature of the error. It is possible that some server processing occurred prior to the error, which could result in undesirable effects if the request were reattempted.

HTTP/2 provides two mechanisms for providing a guarantee to a client that a request has not been processed:

- The GOAWAY frame indicates the highest stream number that might have been processed. Requests on streams with higher numbers are therefore guaranteed to be safe to retry.
- The REFUSED_STREAM error code can be included in a RST_STREAM frame to indicate that the stream is being closed prior to any processing having occurred. Any request that was sent on the reset stream can be safely retried.

Requests that have not been processed have not failed; clients MAY automatically retry them, even those with non-idempotent methods.

A server MUST NOT indicate that a stream has not been processed unless it can guarantee that fact. If frames that are on a stream are passed to the application layer for any stream, then REFUSED_STREAM MUST NOT be used for that stream, and a GOAWAY frame MUST include a stream identifier that is greater than or equal to the given stream identifier.

In addition to these mechanisms, the PING frame provides a way for a client to easily test a connection. Connections that remain idle can become broken as some middleboxes (for instance, network address translators or load balancers) silently discard connection bindings. The PING frame allows a client to safely test whether a connection is still active without sending a request.

HTTP/2 allows a server to pre-emptively send (or "push") responses (along with corresponding "promised" requests) to a client in association with a previous client-initiated request. This can be useful when the server knows the client will need to have those responses available in order to fully process the response to the original request.

A client can request that server push be disabled, though this is negotiated for each hop independently. The SETTINGS_ENABLE_PUSH setting can be set to 0 to indicate that server push is disabled.

Promised requests MUST be cacheable (see [RFC7231], Section 4.2.3), MUST be safe (see [RFC7231], Section 4.2.1), and MUST NOT include a request body. Clients that receive a promised request that is not cacheable, that is not known to be safe, or that indicates the presence of a request body MUST reset the promised stream with a stream error (Section 5.4.2) of type PROTOCOL_ERROR. Note this could result in the promised stream being reset if the client does not recognize a newly defined method as being safe.

Pushed responses that are cacheable (see [RFC7234], Section 3) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present ([RFC7234], Section 5.2.2)) while the stream identified by the promised stream ID is still open.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

The server MUST include a value in the :authority pseudo-header field for which the server is authoritative (see Section 10.1). A client MUST treat a PUSH_PROMISE for which the server is not authoritative as a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

An intermediary can receive pushes from the server and choose not to forward them on to the client. In other words, how to make use of the pushed information is up to that intermediary. Equally, the intermediary might choose to make additional pushes to the client, without any action taken by the server.

A client cannot push. Thus, servers MUST treat the receipt of a PUSH_PROMISE frame as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. Clients MUST reject any attempt to change the SETTINGS_ENABLE_PUSH setting to a value other than 0 by treating the message as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Server push is semantically equivalent to a server responding to a request; however, in this case, that request is also sent by the server, as a PUSH_PROMISE frame.

The PUSH_PROMISE frame includes a header block that contains a complete set of request header fields that the server attributes to the request. It is not possible to push a response to a request that includes a request body.

Pushed responses are always associated with an explicit request from the client. The PUSH_PROMISE frames sent by the server are sent on that explicit request's stream. The PUSH_PROMISE frame also includes a promised stream identifier, chosen from the stream identifiers available to the server (see Section 5.1.1).

The header fields in PUSH_PROMISE and any subsequent CONTINUATION frames MUST be a valid and complete set of request header fields (Section 8.1.2.3). The server MUST include a method in the :method pseudo-header field that is safe and cacheable. If a client receives a PUSH_PROMISE that does not include a complete and valid set of header fields or the :method pseudo-header field identifies a method that is not safe, it MUST respond with a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

The server SHOULD send PUSH_PROMISE (Section 6.6) frames prior to sending any frames that reference the promised responses. This avoids a race where clients issue requests prior to receiving any PUSH_PROMISE frames.

For example, if the server receives a request for a document containing embedded links to multiple image files and the server chooses to push those additional images to the client, sending PUSH_PROMISE frames before the DATA frames that contain the image links ensures that the client is able to see that a resource will be pushed before discovering embedded links. Similarly, if the server pushes responses referenced by the header block (for instance, in Link header fields), sending a PUSH_PROMISE before sending the header block ensures that clients do not request those resources.

PUSH_PROMISE frames MUST NOT be sent by the client.

PUSH_PROMISE frames can be sent by the server in response to any client-initiated stream, but the stream MUST be in either the "open" or "half-closed (remote)" state with respect to the server. PUSH_PROMISE frames are interspersed with the frames that comprise a response, though they cannot be interspersed with HEADERS and CONTINUATION frames that comprise a single header block.

Sending a PUSH_PROMISE frame creates a new stream and puts the stream into the "reserved (local)" state for the server and the "reserved (remote)" state for the client.

After sending the PUSH_PROMISE frame, the server can begin delivering the pushed response as a response (Section 8.1.2.4) on a server-initiated stream that uses the promised stream identifier. The server uses this stream to transmit an HTTP response, using the same sequence of frames as defined in Section 8.1. This stream becomes "half-closed" to the client (Section 5.1) after the initial HEADERS frame is sent.

Once a client receives a PUSH_PROMISE frame and chooses to accept the pushed response, the client SHOULD NOT issue any requests for the promised response until after the promised stream has closed.

If the client determines, for any reason, that it does not wish to receive the pushed response from the server or if the server takes too long to begin sending the promised response, the client can send a RST_STREAM frame, using either the CANCEL or REFUSED_STREAM code and referencing the pushed stream's identifier.

A client can use the SETTINGS_MAX_CONCURRENT_STREAMS setting to limit the number of responses that can be concurrently pushed by a server. Advertising a SETTINGS_MAX_CONCURRENT_STREAMS value of zero disables server push by preventing the server from creating the necessary streams. This does not prohibit a server from sending PUSH_PROMISE frames; clients need to reset any promised streams that are not wanted.

Clients receiving a pushed response MUST validate that either the server is authoritative (see Section 10.1) or the proxy that provided the pushed response is configured for the corresponding request. For example, a server that offers a certificate for only the example.com DNS-ID or Common Name is not permitted to push a response for https://www.example.org/doc.

The response for a PUSH_PROMISE stream begins with a HEADERS frame, which immediately puts the stream into the "half-closed (remote)" state for the server and "half-closed (local)" state for the client, and ends with a frame bearing END_STREAM, which places the stream in the "closed" state.

- Note: The client never sends a frame with the END_STREAM flag for a server push.

In HTTP/1.x, the pseudo-method CONNECT ([RFC7231], Section 4.3.6) is used to convert an HTTP connection into a tunnel to a remote host. CONNECT is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with https resources.

In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes. The HTTP header field mapping works as defined in Section 8.1.2.3 ("Request Pseudo-Header Fields"), with a few differences. Specifically:

- The :method pseudo-header field is set to CONNECT.
- The :scheme and :path pseudo-header fields MUST be omitted.
- The :authority pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests (see [RFC7230], Section 5.3)).

A CONNECT request that does not conform to these restrictions is malformed (Section 8.1.2.6).

A proxy that supports CONNECT establishes a TCP connection [TCP] to the server identified in the :authority pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6.

After the initial HEADERS frame sent by each peer, all subsequent DATA frames correspond to data sent on the TCP connection. The payload of any DATA frames sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is assembled into DATA frames by the proxy. Frame types other than DATA or stream management frames (RST_STREAM, WINDOW_UPDATE, and PRIORITY) MUST NOT be sent on a connected stream and MUST be treated as a stream error (Section 5.4.2) if received.

The TCP connection can be closed by either peer. The END_STREAM flag on a DATA frame is treated as being equivalent to the TCP FIN bit. A client is expected to send a DATA frame with the END_STREAM flag set after receiving a frame bearing the END_STREAM flag. A proxy that receives a DATA frame with the END_STREAM flag set sends the attached data with the FIN bit set on the last TCP segment. A proxy that receives a TCP segment with the FIN bit set sends a DATA frame with the END_STREAM flag set. Note that the final TCP segment or DATA frame could be empty.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error (Section 5.4.2) of type CONNECT_ERROR. Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the HTTP/2 connection.

This section outlines attributes of the HTTP protocol that improve interoperability, reduce exposure to known security vulnerabilities, or reduce the potential for implementation variation.

HTTP/2 connections are persistent. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Clients SHOULD NOT open more than one HTTP/2 connection to a given host and port pair, where the host is derived from a URI, a selected alternative service [ALT-SVC], or a configured proxy.

A client can create additional connections as replacements, either to replace connections that are near to exhausting the available stream identifier space (Section 5.1.1), to refresh the keying material for a TLS connection, or to replace connections that have encountered errors (Section 5.4.1).

A client MAY open multiple connections to the same IP address and TCP port using different Server Name Indication [TLS-EXT] values or to provide different TLS client certificates but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the transport-layer TCP connection, the terminating endpoint SHOULD first send a GOAWAY (Section 6.8) frame so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

Connections that are made to an origin server, either directly or through a tunnel created using the CONNECT method (Section 8.3), MAY be reused for requests with multiple different URI authority components. A connection can be reused as long as the origin server is authoritative (Section 10.1). For TCP connections without TLS, this depends on the host having resolved to the same IP address.

For https resources, connection reuse additionally depends on having a certificate that is valid for the host in the URI. The certificate presented by the server MUST satisfy any checks that the client would perform when forming a new TLS connection for the host in the URI.

An origin server might offer a certificate with multiple subjectAltName attributes or names with wildcards, one of which is valid for the authority in the URI. For example, a certificate with a subjectAltName of *.example.com might permit the use of the same connection for requests to URIs starting with https://a.example.com/ and https://b.example.com/.

In some deployments, reusing a connection for multiple origins can result in requests being directed to the wrong origin server. For example, TLS termination might be performed by a middlebox that uses the TLS Server Name Indication (SNI) [TLS-EXT] extension to select an origin server. This means that it is possible for clients to send confidential information to servers that might not be the intended target for the request, even though the server is otherwise authoritative.

A server that does not wish clients to reuse connections can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request (see Section 9.1.2).

A client that is configured to use a proxy over HTTP/2 directs requests to that proxy through a single connection. That is, all requests sent via a proxy reuse the connection to the proxy.

The 421 (Misdirected Request) status code indicates that the request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

Clients receiving a 421 (Misdirected Request) response from a server MAY retry the request -- whether the request method is idempotent or not -- over a different connection. This is possible if a connection is reused (Section 9.1.1) or if an alternative service is selected [ALT-SVC].

This status code MUST NOT be generated by proxies.

A 421 response is cacheable by default, i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

Implementations of HTTP/2 MUST use TLS version 1.2 [TLS12] or higher for HTTP/2 over TLS. The general TLS usage guidance in [TLSBCP] SHOULD be followed, with some additional restrictions that are specific to HTTP/2.

The TLS implementation MUST support the Server Name Indication (SNI) [TLS-EXT] extension to TLS. HTTP/2 clients MUST indicate the target domain name when negotiating TLS.

Deployments of HTTP/2 that negotiate TLS 1.3 or higher need only support and use the SNI extension; deployments of TLS 1.2 are subject to the requirements in the following sections. Implementations are encouraged to provide defaults that comply, but it is recognized that deployments are ultimately responsible for compliance.

This section describes restrictions on the TLS 1.2 feature set that can be used with HTTP/2. Due to deployment limitations, it might not be possible to fail TLS negotiation when these restrictions are not met. An endpoint MAY immediately terminate an HTTP/2 connection that does not meet these TLS requirements with a connection error (Section 5.4.1) of type INADEQUATE_SECURITY.

A deployment of HTTP/2 over TLS 1.2 MUST disable compression. TLS compression can lead to the exposure of information that would not otherwise be revealed [RFC3749]. Generic compression is unnecessary since HTTP/2 provides compression features that are more aware of context and therefore likely to be more appropriate for use for performance, security, or other reasons.

A deployment of HTTP/2 over TLS 1.2 MUST disable renegotiation. An endpoint MUST treat a TLS renegotiation as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. Note that disabling renegotiation can result in long-lived connections becoming unusable due to limits on the number of messages the underlying cipher suite can encipher.

An endpoint MAY use renegotiation to provide confidentiality protection for client credentials offered in the handshake, but any renegotiation MUST occur prior to sending the connection preface. A server SHOULD request a client certificate if it sees a renegotiation request immediately after establishing a connection.

This effectively prevents the use of renegotiation in response to a request for a specific protected resource. A future specification might provide a way to support this use case. Alternatively, a server might use an error (Section 5.4) of type HTTP_1_1_REQUIRED to request the client use a protocol that supports renegotiation.

Implementations MUST support ephemeral key exchange sizes of at least 2048 bits for cipher suites that use ephemeral finite field Diffie-Hellman (DHE) [TLS12] and 224 bits for cipher suites that use ephemeral elliptic curve Diffie-Hellman (ECDHE) [RFC4492]. Clients MUST accept DHE sizes of up to 4096 bits. Endpoints MAY treat negotiation of key sizes smaller than the lower limits as a connection error (Section 5.4.1) of type INADEQUATE_SECURITY.

A deployment of HTTP/2 over TLS 1.2 SHOULD NOT use any of the cipher suites that are listed in the cipher suite black list (Appendix A).

Endpoints MAY choose to generate a connection error (Section 5.4.1) of type INADEQUATE_SECURITY if one of the cipher suites from the black list is negotiated. A deployment that chooses to use a black-listed cipher suite risks triggering a connection error unless the set of potential peers is known to accept that cipher suite.

Implementations MUST NOT generate this error in reaction to the negotiation of a cipher suite that is not on the black list. Consequently, when clients offer a cipher suite that is not on the black list, they have to be prepared to use that cipher suite with HTTP/2.

The black list includes the cipher suite that TLS 1.2 makes mandatory, which means that TLS 1.2 deployments could have non-intersecting sets of permitted cipher suites. To avoid this problem causing TLS handshake failures, deployments of HTTP/2 that use TLS 1.2 MUST support TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 [TLS-ECDHE] with the P-256 elliptic curve [FIPS186].

Note that clients might advertise support of cipher suites that are on the black list in order to allow for connection to servers that do not support HTTP/2. This allows servers to select HTTP/1.1 with a cipher suite that is on the HTTP/2 black list. However, this can result in HTTP/2 being negotiated with a black-listed cipher suite if the application protocol and cipher suite are independently selected.

HTTP/2 relies on the HTTP/1.1 definition of authority for determining whether a server is authoritative in providing a given response (see [RFC7230], Section 9.1). This relies on local name resolution for the "http" URI scheme and the authenticated server identity for the "https" scheme (see [RFC2818], Section 3).

In a cross-protocol attack, an attacker causes a client to initiate a transaction in one protocol toward a server that understands a different protocol. An attacker might be able to cause the transaction to appear as a valid transaction in the second protocol. In combination with the capabilities of the web context, this can be used to interact with poorly protected servers in private networks.

Completing a TLS handshake with an ALPN identifier for HTTP/2 can be considered sufficient protection against cross-protocol attacks. ALPN provides a positive indication that a server is willing to proceed with HTTP/2, which prevents attacks on other TLS-based protocols.

The encryption in TLS makes it difficult for attackers to control the data that could be used in a cross-protocol attack on a cleartext protocol.

The cleartext version of HTTP/2 has minimal protection against cross-protocol attacks. The connection preface (Section 3.5) contains a string that is designed to confuse HTTP/1.1 servers, but no special protection is offered for other protocols. A server that is willing to ignore parts of an HTTP/1.1 request containing an Upgrade header field in addition to the client connection preface could be exposed to a cross-protocol attack.

The HTTP/2 header field encoding allows the expression of names that are not valid field names in the Internet Message Syntax used by HTTP/1.1. Requests or responses containing invalid header field names MUST be treated as malformed (Section 8.1.2.6). An intermediary therefore cannot translate an HTTP/2 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/2 allows header field values that are not valid. While most of the values that can be encoded will not alter header field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a header field value MUST be treated as malformed (Section 8.1.2.6). Valid characters are defined by the field-content ABNF rule in Section 3.2 of [RFC7230].

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server MUST ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Pushed responses for which an origin server is not authoritative (see Section 10.1) MUST NOT be used or cached.

An HTTP/2 connection can demand a greater commitment of resources to operate than an HTTP/1.1 connection. The use of header compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH_PROMISE frames is not constrained in the same fashion. A client that accepts server push SHOULD limit the number of streams it allows to be in the "reserved (remote)" state. An excessive number of server push streams can be treated as a stream error (Section 5.4.2) of type ENHANCE_YOUR_CALM.

Processing capacity cannot be guarded as effectively as state capacity.

The SETTINGS frame can be abused to cause a peer to expend additional processing time. This might be done by pointlessly changing SETTINGS parameters, setting multiple undefined parameters, or changing the same setting multiple times in the same frame. WINDOW_UPDATE or PRIORITY frames can be abused to cause an unnecessary waste of resources.

Large numbers of small or empty frames can be abused to cause a peer to expend time processing frame headers. Note, however, that some uses are entirely legitimate, such as the sending of an empty DATA or CONTINUATION frame at the end of a stream.

Header compression also offers some opportunities to waste processing resources; see Section 7 of [COMPRESSION] for more details on potential abuses.

Limits in SETTINGS parameters cannot be reduced instantaneously, which leaves an endpoint exposed to behavior from a peer that could exceed the new limits. In particular, immediately after establishing a connection, limits set by a server are not known to clients and could be exceeded without being an obvious protocol violation.

All these features -- i.e., SETTINGS changes, small frames, header compression -- have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that doesn't monitor this behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error (Section 5.4.1) of type ENHANCE_YOUR_CALM.

A large header block (Section 4.3) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header block, which prevents streaming of header fields to their ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean that an endpoint might need to buffer the entire header block. Since there is no hard limit to the size of a header block, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the SETTINGS_MAX_HEADER_LIST_SIZE to advise peers of limits that might apply on the size of header blocks. This setting is only advisory, so endpoints MAY choose to send header blocks that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to a connection, so any request or response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger header block than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code [RFC6585]. A client can discard responses that it cannot process. The header block MUST be processed to ensure a consistent connection state, unless the connection is closed.

The CONNECT method can be used to create disproportionate load on an proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME_WAIT state. Therefore, a proxy cannot rely on SETTINGS_MAX_CONCURRENT_STREAMS alone to limit the resources consumed by CONNECT requests.

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/2 enables compression of header fields (Section 4.3); the following concerns also apply to the use of HTTP compressed content-codings ([RFC7231], Section 3.1.2.1).

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression dictionaries are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined. Generic stream compression, such as that provided by TLS, MUST NOT be used with HTTP/2 (see Section 9.2).

Further considerations regarding the compression of header fields are described in [COMPRESSION].

Padding within HTTP/2 is not intended as a replacement for general purpose padding, such as might be provided by TLS [TLS12]. Redundant padding could even be counterproductive. Correct application can depend on having specific knowledge of the data that is being padded.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Use of padding can result in less protection than might seem immediately obvious. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

Intermediaries SHOULD retain padding for DATA frames but MAY drop padding for HEADERS and PUSH_PROMISE frames. A valid reason for an intermediary to change the amount of padding of frames is to improve the protections that padding provides.

Several characteristics of HTTP/2 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the manner in which flow-control windows are managed, the way priorities are allocated to streams, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client, as defined in Section 1.8 of [HTML5].

HTTP/2's preference for using a single TCP connection allows correlation of a user's activity on a site. Reusing connections for different origins allows tracking across those origins.

Because the PING and SETTINGS frames solicit immediate responses, they can be used by an endpoint to measure latency to their peer. This might have privacy implications in certain scenarios.

A string for identifying HTTP/2 is entered into the "Application-Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

This document establishes a registry for frame types, settings, and error codes. These new registries appear in the new "Hypertext Transfer Protocol version 2 (HTTP/2) Parameters" section.

This document registers the HTTP2-Settings header field for use in HTTP; it also registers the 421 (Misdirected Request) status code.

This document registers the PRI method for use in HTTP to avoid collisions with the connection preface (Section 3.5).

This document creates two registrations for the identification of HTTP/2 (see Section 3.3) in the "Application-Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

The "h2" string identifies HTTP/2 when used over TLS:

Protocol:

HTTP/2 over TLS

Identification Sequence:

0x68 0x32 ("h2")

Specification:

This document The "h2c" string identifies HTTP/2 when used over cleartext TCP:

Protocol:

HTTP/2 over TCP

Identification Sequence:

0x68 0x32 0x63 ("h2c")

Specification:

This document

This document establishes a registry for HTTP/2 frame type codes. The "HTTP/2 Frame Type" registry manages an 8-bit space. The "HTTP/2 Frame Type" registry operates under either of the "IETF Review" or "IESG Approval" policies [RFC5226] for values between 0x00 and 0xef, with values between 0xf0 and 0xff being reserved for Experimental Use.

New entries in this registry require the following information:

Frame Type:

A name or label for the frame type.

Code:

The 8-bit code assigned to the frame type.

Specification:

A reference to a specification that includes a description of the frame layout, its semantics, and flags that the frame type uses, including any parts of the frame that are conditionally present based on the value of flags.

The entries in the following table are registered by this document.

Frame	Type Code	Section
DATA	0x0	Section 6.1
HEADERS	0x1	Section 6.2
PRIORITY	0x2	Section 6.3
RST_STREAM	0x3	Section 6.4
SETTINGS	0x4	Section 6.5
PUSH_PROMISE	0x5	Section 6.6
PING	0x6	Section 6.7
GOAWAY	0x7	Section 6.8
WINDOW_UPDATE	0x8	Section 6.9
CONTINUATION	0x9	Section 6.10

This document establishes a registry for HTTP/2 settings. The "HTTP/2 Settings" registry manages a 16-bit space. The "HTTP/2 Settings" registry operates under the "Expert Review" policy [RFC5226] for values in the range from 0x0000 to 0xefff, with values between and 0xf000 and 0xffff being reserved for Experimental Use.

New registrations are advised to provide the following information:

Name:

A symbolic name for the setting. Specifying a setting name is optional.

Code:

The 16-bit code assigned to the setting.

Initial Value:

An initial value for the setting.

Specification: An optional reference to a specification that describes the use of the setting.

The entries in the following table are registered by this document.

Name	Code	Initial Value	Specification
HEADER_TABLE_SIZE	0x1	4096	Section 6.5.2
ENABLE_PUSH	0x2	1	Section 6.5.2
MAX_CONCURRENT_STREAMS	0x3	(infinite)	Section 6.5.2
INITIAL_WINDOW_SIZE	0x4	65535	Section 6.5.2
MAX_FRAME_SIZE	0x5	16384	Section 6.5.2
MAX_HEADER_LIST_SIZE	0x6	(infinite)	Section 6.5.2

This document establishes a registry for HTTP/2 error codes. The "HTTP/2 Error Code" registry manages a 32-bit space. The "HTTP/2 Error Code" registry operates under the "Expert Review" policy [RFC5226].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Name:

A name for the error code. Specifying an error code name is optional.

Code:

The 32-bit error code value.

Description:

A brief description of the error code semantics, longer if no detailed specification is provided.

Specification:

An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

Name	Code	Description	Specification
NO_ERROR	0x0	Graceful shutdown	Section 7
PROTOCOL_ERROR	0x1	Protocol error detected	Section 7
INTERNAL_ERROR	0x2	Implementation fault	Section 7
FLOW_CONTROL_ERROR	0x3	Flow-control limits exceeded	Section 7
SETTINGS_TIMEOUT	0x4	Settings not acknowledged	Section 7
STREAM_CLOSED	0x5	Frame received for closed stream	Section 7
FRAME_SIZE_ERROR	0x6	Frame size incorrect	Section 7
REFUSED_STREAM	0x7	Stream not processed	Section 7
CANCEL	0x8	Stream cancelled	Section 7
COMPRESSION_ERROR	0x9	Compression state not updated	Section 7
CONNECT_ERROR	0xa	TCP connection error for CONNECT method	Section 7
ENHANCE_YOUR_CALM	0xb	Processing capacity exceeded	Section 7
INADEQUATE_SECURITY	0xc	Negotiated TLS parameters not acceptable	Section 7
HTTP_1_1_REQUIRED	0xd	Use HTTP/1.1 for the request	Section 7

This section registers the HTTP2-Settings header field in the "Permanent Message Header Field Names" registry [BCP90].

Header field name:

HTTP2-Settings

Applicable protocol:

http

Status:

standard

Author/Change controller:

IETF

Specification document(s):

Section 3.2.1 of this document

Related information:

This header field is only used by an HTTP/2 client for Upgrade-based negotiation.

This section registers the PRI method in the "HTTP Method Registry" ([RFC7231], Section 8.1).

Method Name:

PRI

Safe

Yes

Idempotent

Yes

Specification document(s)

Section 3.5 of this document

Related information:

This method is never used by an actual client. This method will appear to be used when an HTTP/1.1 server or intermediary attempts to parse an HTTP/2 connection preface.

This document registers the 421 (Misdirected Request) HTTP Status code in the "HTTP Status Codes" registry" ([RFC7231], Section 8.2).

Status Code:

421

Short Description:

Misdirected Request

Specification:

Section 9.1.2 of this document

This document registers the "h2c" upgrade token in the "Hypertext Transfer Protocol (HTTP) Upgrade Token Registry" registry ([RFC7230], Section 8.6).

Value:

h2c

Description:

Hypertext Transfer Protocol version 2 (HTTP/2)

Expected Version Tokens:

None

Reference:

Section 3.2 of this document